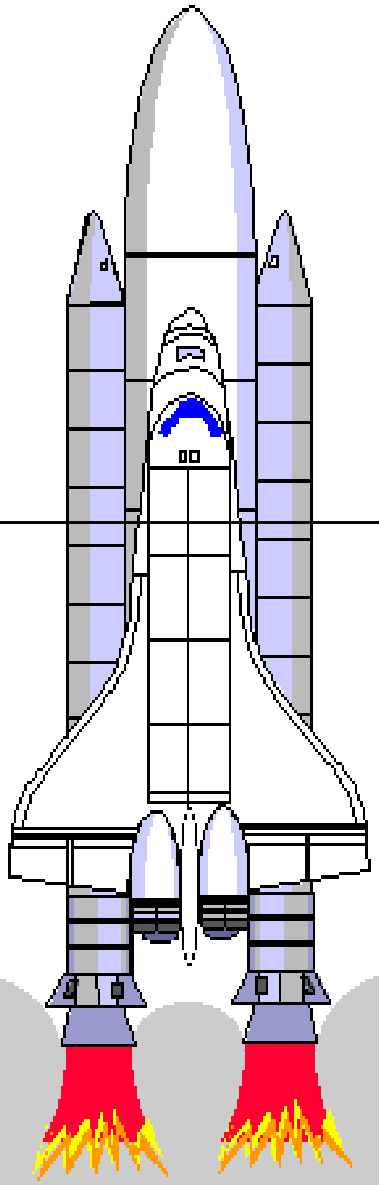
HOOD USER MANUAL

RELEASE 1.0
HOOD USERs GROUP



HOOD USER'sMANUAL
Issue: **Working Document 1.0**
Created: July 27, 1994
Reference: **HUM3-1**
Prepared by: **HUM Working Group**
Approved by: **HOOD Technical Group**
Diffusion **HOOD USERs Group**

Doc.Ref.: HUM-1.0

Printed : July 11, 1996 1:38 pm

HOOD USER MANUAL

Technical Document property of the HUG

Document No.: HUM

Issue No.: 1.0

Last Modified : July 11, 1996 1:38 pm

Abstract : HOOD (**H**ierarchical **O**bject **O**riented **D**esign, a trademark of The Hood User Group) is the industrial design method chosen by the European Space Agency (ESA) as the common method for European software projects. HOOD is a design method supporting the definition of interfaces, reusable modules as well as architecture models. It allows to represent systems and software architecture as a set of interconnected hierarchies of objects.

HOOD is therefore supporting programming in the large, relying on code generator technology from high level or formal notations. HOOD is thus primarily aiming to better fill the needs of the prime contractor and integrator than those of the low level programmer. Although HOOD puts the emphasis on interface and behaviour mastering, these benefits have been rather neglected by today users of graphical programming notations for object oriented programming languages.

This document presents the HOOD development approach and gives examples for both mastering architecture definition and integration of components developed with different technologies and teams in the area of large and complex, real time, critical and distributed data processing systems.

Index Terms: Method, Integration, HOOD, Object, Design, Formal, Verification, Control Expression, Real-Time, Reliability, Distribution, Test.

PREFACE

The HOOD method was developed in 1987 under European Space Agency (ESA) contract A0/1-1890/86/NL/MA by a consortium of Cisi, CRI A/S and Matra Marconi Space. **HOOD™ is a registered Trademark of the HOOD User Group**. This fact must be stated in any publication referencing the name of HOOD in the context of the HOOD method as the basis of the publication.
HOOD has been selected by ESA projects including Columbus and Hermes as the design method for the Architectural Design phase. Since, HOOD was selected by several large long lived project from aerospace and industry.

In 1991, the HOOD USER GROUP (HUG) was setup as a non profit organisation aiming to provide support for sharing experience and to control the evolution of the method. The HUG is organized in a STEERING GROUP (HSG) in charge of administrative issues, and in a TECHNICAL GROUP (HTG) in charge of all technical issues possibly delegating work to specific WORKING GROUPs.
The HUG is officially based at:

C/O Spacebel Informatique, attn HUG
11,rue Colonel Bourg
B-1140 BRUSSELS, Belgium
Tel.: (32) 2.730.46.50
Fax: (32).2.726.85.13
e-mail: hug@spacebel.be
http://www.spacebel.be

The HOOD REFERENCE MANUAL Issue 3.1 was developed in 1991 by the HOOD TECHNICAL GROUP and approved for two years by vote by the HOOD USER GROUP at the Pisa (Italy) april 3rd 1992 HUG meeting, and published by MASSON and PRENTICE HALL in 1993.

The HOOD Manual Issue 3.0 has been further developed in 1989 in response to user experience by the HOOD Working Group comprising representatives of ESTEC, COLUMBUS and HERMES projects.

The present document is the HOOD USER'S MANUAL and has as goals to illustrate how to use the HOOD method for the design and development of large, complex and reliable software.

The material of this book shows usage of HOOD in various application domains, but avoids too much arguing on how the thing is done. The authors have tried to leave each part self documented, so that there is no precise rule to read the book, and that readers can jump from one chapter to another according to their present needs on information about HOOD usage. There is however a general line followed by the contents:

- chapter one is a summary of the method concepts, including some terminology, and some HOOD examples and illustrations,
- chapter two describes the HOOD ideal usage in the ideal project: you have time, man-power, and tools. It gives advanced hints and techniques on how to produce HOOD designs and models in several common areas in the data processing domain. These hints are given from a pure technical point of view, ignoring any project and support environment constraints,
- chapter three tries to help HOOD designers trapped in the constraints of a particular context and project. It gives advice on how to cope and integrating HOOD techniques within given

constraints, standards and toolsets.

This book is not a HOOD study, nor a pedagogic manual; we hope that despite of the different backgrounds of the readers, they will manage to understand the HOOD spirit that the authors have tried to put down along the different sections.

The current version is a working document intended only to be distributed among HOOD User Group members only .
It has to be considered as set of technical notes providing a synthesis of the industrial experience of the authors and various contributors.

A further version will take into account new concepts introduced within the HOOD4 evolution.

OBSERVATION REPORTS

For submission of comments for modification or extension to this manual, we would appreciate them being sent via e-mail to the following address:

heitz@cisi.cnes.fr

If you do not have e-mail access, please send the comments using the special Observation Report (OR) form given on the following page, to HOOD TECHNICAL GROUP at the following address:

Attn HTG-Chairman
Maurice HEITZ,
CISI
13, rue Villet
31400 TOULOUSE - FRANCE
fax 33 - 61.17.66.96

Observation report FORM

OBSERVATION REPORT (OR) From			
Document under review: *HOOD USER MANUAL*	**Para:**	**Page:**	**No:**
Subject: **Discrepancy:**			
Recommendation:			

ACKNOWLEDGMENTS

The assistance provided for meetings of the HOOD TECHNICAL GROUP by CISI, CRI AS, ESA-ESTEC, INTECS, LOGICA, MATRA MARCONI SPACE, and SEMA GROUP, is gratefully acknowledged : without their support this work could not have been undertaken. We are also grateful for the support of the employers of the HOOD TECHNICAL GROUP members and other organisations that provided valuable feedbacks for the HOOD definition.

Particular thanks are due to MATRA MARCONI SPACE, CISI and CNES whose support made possible the production of this book. A number of ideas and advices have been borrowed from studies funded by the ESPRIT or EUREKA programmes and projects. We gratefully acknowledge the work done in the TOOL'USE, COMPLEMENT, ESF and PROTEUS projects. Other significant inputs were taken from HOOD training course materials developed by CISI and MMS.

On a more personal note, we are especially grateful to the authors of observation reports on earlier definition of the HOOD method, and to all members of the HOOD TECHNICAL GROUP who contributed with many technical notes and to multiple and detailed discussions. Special thanks for contributions from Giancarlo SAVOIA, Rainer GERLICH, Peter J ROBINSON and Andy CARMICHAEL.
The authors want also to thank all early readers for their comments and discussions which contributed significantly on the elaboration of this manual.

Attendees of HOOD TECHNICAL GROUP included :

Edouard ANDRE	[Sema Group]	
Jorge AMADOR	[ESA/ESTEC]	
Andy CARMICHAEL	[Systematica]	
Javier CAMPOS	[GMV]	
Bernard DELATTE	[CNES]	
Jean Marie WALLUT	[CNES]	
Pierre DISSAUX	[TNI]	
Antony ELLIOT	[IPSYS]	
Rainer GERLICH	[Dornier]	
Winfried BOELKE	[ERNO]	
Patrice Micouin	[Steria]	
Alain Paul Andry	[Trasys]	
Maurice HEITZ	[Cisi]	(HTG chairman)
Jean François MULLER	[MMS]	(HTG secretary)
Christophe .PINAUD	[MMS]	
Peter J ROBINSON	[CANA]	
Giancarlo SAVOIA	[Intecs]	

We also acknowledge gratefully the support of Jardine BARRINGTON-COOK, chairman of the HOOD STEERING GROUP, and the HSG secretariesPatrick Van der DONCKTand Guy PAQUET, who provided the necessary liaison between the HOOD TECHNICAL GROUP and the HOOD USERs GROUP.

B. DELATTE, M. HEITZ, J.F. MULLER,C.PINAUD Editors

About the Authors

Bernard Delatte is an engineer in computer science and is responsible for Object Oriented methods support at **CNES**, 18 av Edouard Belin, 31055 TOULOUSE, FRANCE, Phone: (33) 61.27.49.51 Fax: (33) 61.27.30.84 **e-mail: bdelatte@cst.cnes.fr**
Bernard joined CNES in 1984 as a software quality assurance engineer on the SPOT1 project, and then moved onto software engineering, Ada and HOOD support activities.
Bernard is an active member of the HOOD TECHNICAL GROUP and is publisher of the HOOD NEWS letters (2 issues a year). He is also involved in introducing OOA methods into space software developments and is conducting research activities on the co-operation of OOA methods and HOOD.

Maurice Heitz is an engineer in energetics and mechanics and is consultant on software engineering within the **CISI GROUP** 13, rue Villet, 31400 TOULOUSE, Phone: (33) 61.17.66.66, Fax (33).61.17.66.96.**e-mail heitz@cisi.cnes.fr**. Since 1983 Maurice developed the company's Ada education programme, and was technical manager of several Ada projects for the French National Space Agency and the European Space Agency, before working on formal methods within ESPRIT projects. He is currently involved in system engineering, software engineering, research, development and training activities.
Maurice was one of the lead designers for the definition of HOOD, and he is now an active member and chairman of the HOOD TECHNICAL GROUP. Maurice provided several contributions to Ada Europe and software engineering conferences about HOOD and its application to distributed, fault tolerant systems, reuse and integration with formal methods.

Jean François Muller is an engineer in automatics and electronics and is responsible for the definition of software engineering tools at **MATRA MARCONI SPACE** 1,rue des Cosmonautes, 31400 TOULOUSE, Phone: (33) 61.39.68.12, Fax (33).62.24.77.80 **e-mail muller@soleil.matra_espace.fr**. Since 1982, Jean François was responsible for the development of the HIPPARCOS on board software OS, and later for the COLUMBUS Software Development Environment. Jean François was then project leader for the EUREKA ESF/FERESA and the space FIP subprojects dealing with distributed software developments. Jean François was one of the codesigners of HOOD, and he is now an active member of the HOOD TECHNICAL GROUP.

Christophe Pinaud is a software engineer and is responsible for the definition and use of the software engineering methods at **MATRA MARCONI SPACE** 1,rue des Cosmonautes, 31400 TOULOUSE, Phone: (33) 61.39.65.40, Fax (33).62.24.77.80 **e-mail pinaud-christophe@mms.matra_espace.fr**. Since 1987, Christophe has worked on software engineering tools and methods evaluation and application and on Man Machine Interface development methods and tools. He then participated to the Hermes Software Development Environment definition and development and has supported more than thirteen projects using HOOD (for on-board and ground systems using mainly Ada and C langages) from the training to the design and implementation within toolsets.He is now an active member of the HOOD TECHNICAL GROUP.

LIST of FIGURES

LIST of FIGURES

LIST of FIGURES

1 GETTING STARTED

1.1 GENERAL PRESENTATION

HOOD™ (**H**ierarchical **O**bject **O**riented **D**esign) is a design method, which is used after the requirement analysis activities and covers architectural design, detailed design and coding. The method resulted from merging methods known as *abstract machines* and *object oriented design* and was further adapted to the needs of European Software Industry as an attempt to unify and integrate object orientation and advanced software engineering concepts and notations*[2]*.

1.1.1 HISTORY AND OBJECTIVES

In September 1989, HOOD3.0 was released by the **HOOD Technical Group** founded by ESA. In July 1992 an evolution of the design method **HOOD, release 3.1,** incorporating feed-back over two years experience on various projects, was adopted by the **H**OOD **U**ser's **G**roup (HUG) *[4]*as the method official release see HRM*[1]*.

After an evaluation phase on small pilot projects, the method was chosen for the COLUMBUS Manned Space and ARIANE5 programs. Since, it has been adopted by EUROCOPTER, the French NAVY and by several other large projects in aerospace, defence, transport, energy and nuclear applications.

HOOD is an *object based method* that supports a modular technology, centered on client-server and composition relationships, where minor emphasis is put on inheritance. This is why, beside the recent blossoming of Object Oriented methods, HOOD has evolved slowly and a new release is only planned to be compatible with Ada95 *[7]*. Experience has proven that inheritance in requirement analysis is not the same as inheritance used for implementation, thus there is a need for an intermediate method in design. The advantages promised by *full inheritance based* object oriented methods are currently not balancing the drawbacks of introducing them straight on large scale projects. Furthermore we are convinced that industrial use of inheritance will be limited to data modelling and detailed design programming support. Today OOP is only made possible through powerful programming and debugging support environments, and limited to very small integrated teams. HOOD on the other side has been recently worked out to integrate both modular and inheritance programming, thus making HOOD the architectural design method of choice.

With more than thousand engineers trained in Europe and the availability of several toolsets and companies providing support for using the method, HOOD is spreading continuously within the industry. The HUG*[4]* has been set-up as an international non profit organisation and is in charge of controlling the method evolution.

1.1.2 HOOD IN THE DEVELOPMENT ACTIVITIES

HOOD supports identification of a software architecture after requirement analysis activities and leads naturally into detailed design where operations of objects are further designed using a Program Description Language (PDL) inspired from Ada. This detailed design description may be further refined into target language descriptions up to a point where the target code can be gen-

erated. The figure hereafter (Figure 1 -) indicates HOOD applicability within a simplified life-cycle model.

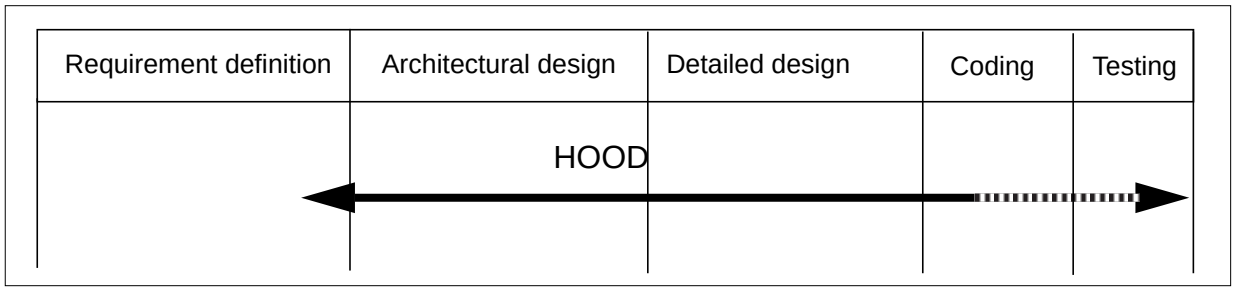


Figure 1 - HOOD in the development activities

Due to the complexity and duration of activities in a large project, HOOD design activities may take place several times in a real development life-cycle. Traceability with requirement analysis as they are refined followed by refined solutions is still a problem in the management of large projects. However HOOD refinement properties provide support for a development approach that goes across the different design phases and helps support consistency of a design solution from the initial approved solution down to the implementation one, and that is still specific and adapted to particular target systems.

These issues may be more refined in dedicated sections of this document (see Section 2.1.2, or Section 2.1.3) but what the reader should already point out now is that:

- HOOD is not a requirement analysis method
- HOOD handles “design requirement analysis” activities in transitioning from requirements analysis to design (see Section 1.3 below)
- HOOD is an architectural design method, helping a designer **to partition the software into modules or objects** of well defined interfaces that can either be directly implemented or further partitioned with HOOD into modules of lower complexity.
- HOOD concepts looking for integration of design with the other development activities. Especially, HOOD object properties having been defined *in order to ease interface mastering, testing and integration* in the context of parallel, multi-people team developments.

1.1.3 BASIC CONCEPTS

HOOD is a method based on:

- **a formalism**:
  - the **graphical notation** allows to express an abstraction of a solution in a clear, high level and easy-to-understand notation.
  - the **textual notation** allows formal expression and refinement of the object’s characteristics and properties by means of an **Object Description Skeleton** (**ODS**). This concept helps to structure the descriptions into separate fields which support appropriate control and program description notations. Finally these descriptions may be fully translated into a target language (Ada, C, C++, or Fortran).
- **a process** describing and refining a software model from abstract structures and concepts towards target machine code.

- **rules** coming from industrial experiences.

These notations allow to use powerful structuring concepts for describing and organising a system as a set of interconnected hierarchies of objects. The principles supported are threefold:

- the notations allow hierarchic representation supporting both global and local views of components. Furthermore the concept of uncle and environment object supports formal representation of a *context of a model* with full consistency checking.
- the graphical notation is an abstraction of a fully detailed model, offering *reduced but consistent views*, and allowing hierarchical refinement of representations and understanding.
- the textual notations leave provisions for both informal and formal texts, allowing the definition of a documentation skeleton as a framework for a step by step integration of advanced notations to capture and formally verify the characteristics of an object.

1.1.3.1 HOOD Objects

Within HOOD, everything is an object, but the term “object” does have the general meaning of MODULE encapsulating a number of properties. However a number of modules may be characterized more precisely. In order to ease the understanding of the method, one can group objects different into different categories according to three perspectives[1]:

- **decomposition**: characterizes objects according to the way they are structured,
- **behaviour**: characterizes objects according to their behavioural properties,
- **organization**: characterizes objects according to their role in the system development.

A synthetic view of this classification is provided in Figure 2 - below.

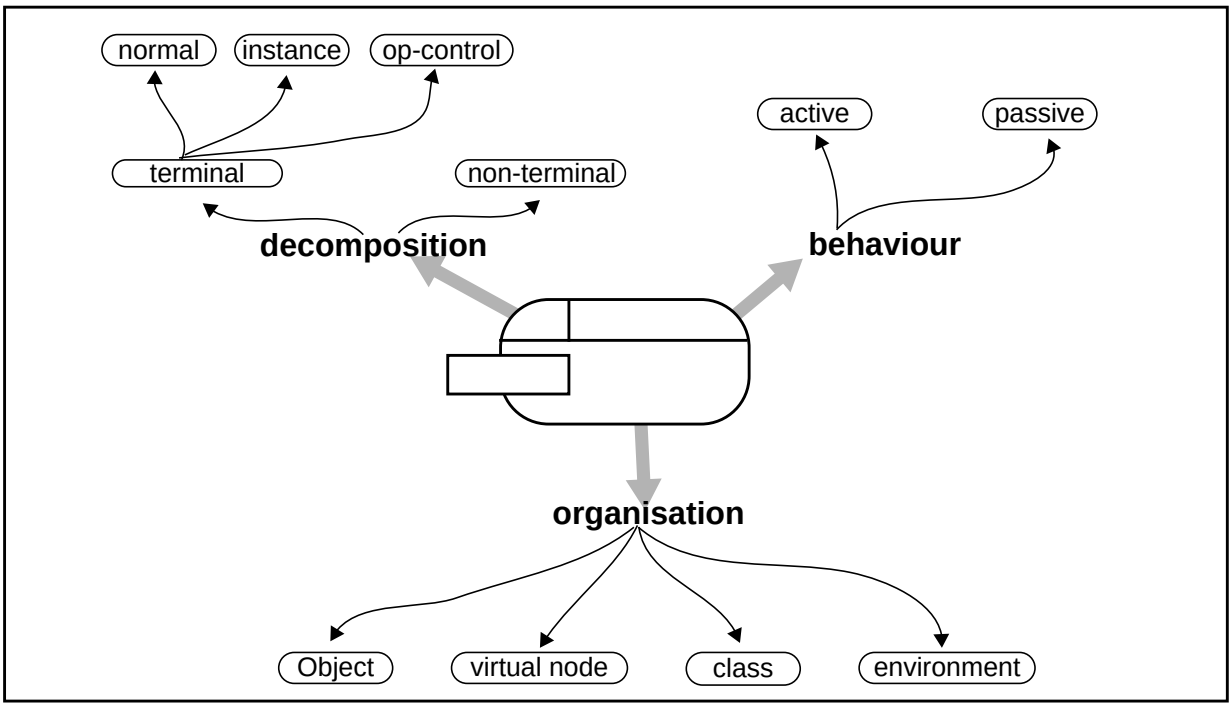


Figure 2 - Classification of the different HOOD Objects

[1]. be careful those views are not facets; in other words, each HOOD object must not be characterized by a term in each of those views, and some of these types are exclusive

In the following, we give a more detailed description of the HOOD object concept as a summary of its description in the HRM*[1]*.
The *object* is the *unit of modularity* and is defined by the services that it *provides*, encapsulating a *state*, and a set of services that it *requires* from other objects. To require a service, a client object must identify the server object to perform it; thus every object has a unique identifier and may be referenced unambiguously. A *service* is an abstraction of an operation or an access to an *attribute / component* of the state of an object.
Each service has a *signature*, (expressed using an Ada like syntax) which defines the number and type of its arguments. The set of signatures constitutes the provided interface; similarly, the required interface comprises the set of signatures of the required services. Communication between objects - the USE relationship - is only possible by service requests, which are similar to Smalltalk method calls or Ada procedure or task entry calls:

- there is a many-to-one connection pattern, reflected by the naming scheme.
- the provider of the service must be named by the client.
- the name of the client is not directly known by the server.
- a client object requesting a service from a server object is said to use the server and this relation is represented by a bold use arrow.
- information items can be exchanged between communicating objects; the associated information flow may be bidirectional and is represented by arrows along the use relationship one (see Figure 3 -)

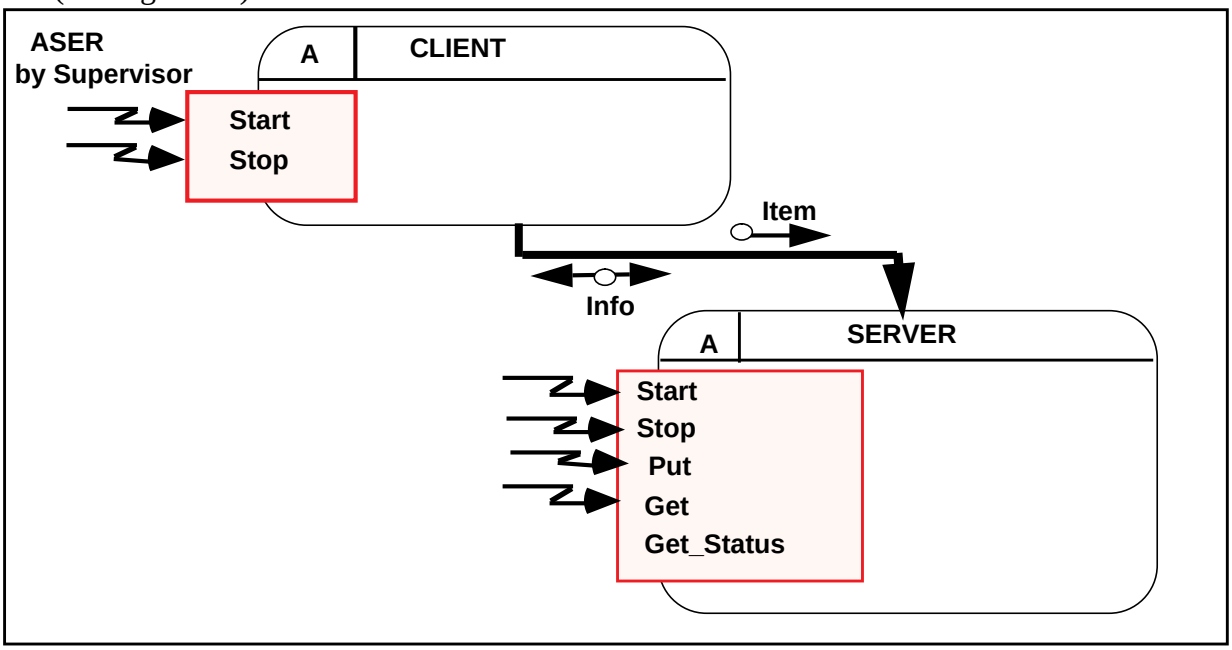


Figure 3 - Objects providing and requiring services and exchanging information items

Communication protocols may be achieved by attaching execution requests and constraints to a given service. Two kinds of constraints are defined:

- *state constraints* (service activation constraints relying on the semantics and internal state of the object) are represented through non labelled trigger arrows.
- *protocol execution constraints* (represented by text labels attached to a trigger arrow) which define that the client requesting process execution is suspended after its request :
  - until full completion of the requested service (Highly Synchronous Execution Request or *HSER protocol*)

- until completion of the request processing by a server process (Loosely Synchronous Execution Request or *LSER protocol)*
- until runoff of a time delay or until full completion of the requested service (*TOER_HSER protocol*)
- until runoff of a time delay or until completion of the request processing by a server process (*TOER_LSER protocol*)
- or not suspended (ASynchronous Execution Request or *ASER protocol)*. This kind of execution request corresponds to classical message passing communication.

1.1.3.2 Control Structures

HOOD allows the precise description of the dynamical behaviour attached to the execution of services (with or without attached execution constraints) through *processes* which are ultimately implemented on target machine processors. The behaviour of processes is captured through two orthogonal concepts:

- **the OPeration Control Structure** (*OPCS, one by service/operation*) which describes how a sequential process executes within an operation.
  Example: For a FILE object with *read* and *write* constraints operations, the OPCS of the *read* operation consists in reading a file contents.
- **the OBject Control Structure** (*OBCS one per object)* which controls the activation and triggering of constrained operations, whereas non constrained operation execute directly under the client requesting process (see Figure 4 -).

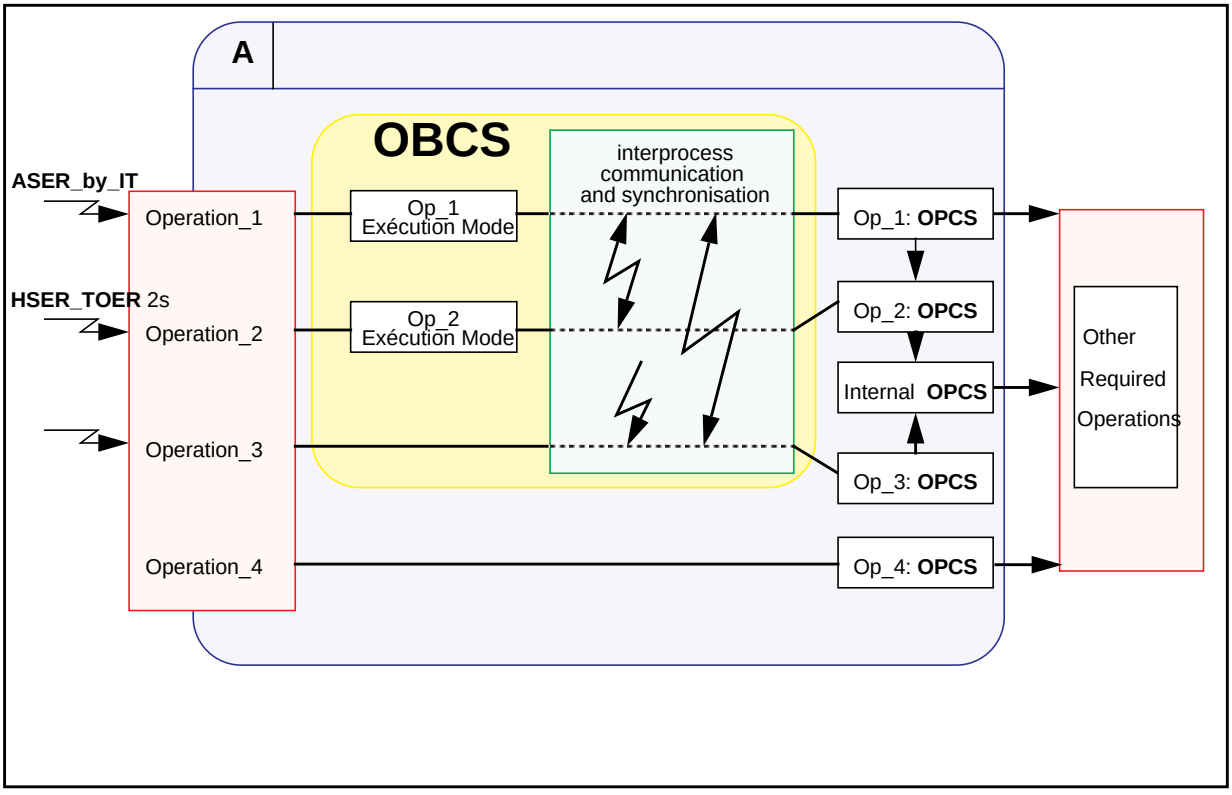


Figure 4 - HOOD Object execution model

This model supports:

- A complete separation of reactive and transformational parts.
- An easy mapping to ADA tasking.
- The possibility of prototyping the dynamics.
- The possibility of expressing the dynamic behaviour using other formalism.
- The definition of execution mode at server level (not at client side)

1.1.3.3 The Include relationship

An object (*called a parent or a non-terminal object)* may itself be defined as the composition of other objects (*called child objects*). Provided services are then *implemented_by* child provided services. The *implemented_by* relationship is represented by a dashed one-to-one arrow.

- *Child objects* are represented as objects within the parent internals, and may require services provided by uncles (objects which are used by their parents) represented as a connection port attached to the parent boundaries. *Terminal objects* are objects that are not further decomposed into child objects.
- *Environment objects* are objects that do not belong to the current parent-child hierarchy and are represented as uncle objects with an “E” in left side

.

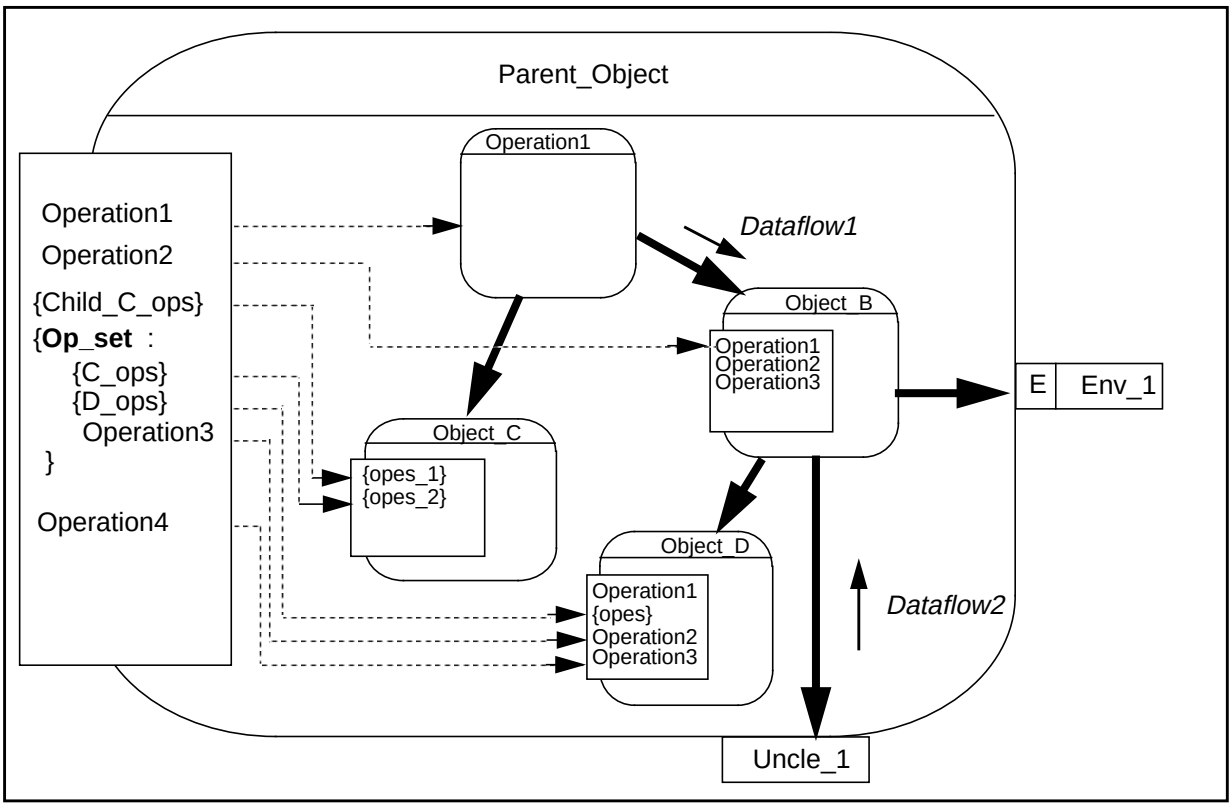


Figure 5 - Representation of Parent, Child Uncle and Environment objects

1.1.3.4 Types, Data and Attributes

Data can only be exchanged between objects which are related by a USE relationship. Data may be:

- either instances of target language types (Ada, C, Fortran or Assemblers)
- or *instances of abstract data types* which are defined in another HOOD object, from which the type definition is then required. Appendix3 gives a detailed illustration of a Stack implementation as a *HOOD Abstract Data Type* (HADT) object.

Types

HOOD enforces a flat typing structure as supported by most target languages. Types are declared either:

- in the *Provided Interface* of the current object. In that case, they must be related to parameters of provided operations. An object should not provide any type which is not an operation parameter type.
- in its *Internals*,
- or they are provided by another object (environment or HADT ones)

Data (or variable)

Data may only be declared in:

- the *Internal* parts of a terminal object. This restriction prohibits the definition of architectures with global data, recognized as poorly reliable and maintainable ones.
- the *local data* part of an OPCS (or OBCS) in which case this data is a temporary one and only exists during the execution of the OPCS.
  For example, a state data shall be declared as much as possible locally within the OBCS in order to not be used by the OPCS.

Attributes

Attributes of an object are data that define invariant properties of an object. While some object models define attributes as a valued data readable by clients, *HOOD unifies all access to an object through operations.* Hence attributes of an object in HOOD are modelled as provided operations (possibly returning an attribute value).

1.1.3.5 Class Objects

A HOOD class object [2]*is defined as a template for objects* which may be parameterized by services (types, constants and/or operations) and allowing to factor the descriptions of identical or similar objects.

[2] The term class object came out as the OOP terminology was not well established. HOOD CLASS object shall not be confused with OOP classes, but are templates or generic objects (similar to generic Ada packages). A HOOD object has always the meaning of “module”. **In order to avoid misunderstanding with OOP terminology the terms “class instance” should always be used to designate an OOP object instance**.

A class object is defined as a separate *root object* (top-level object of a parent-child hierarchy), and may itself be decomposed only into objects and/or instances of other class objects. (see Figure 6 -)

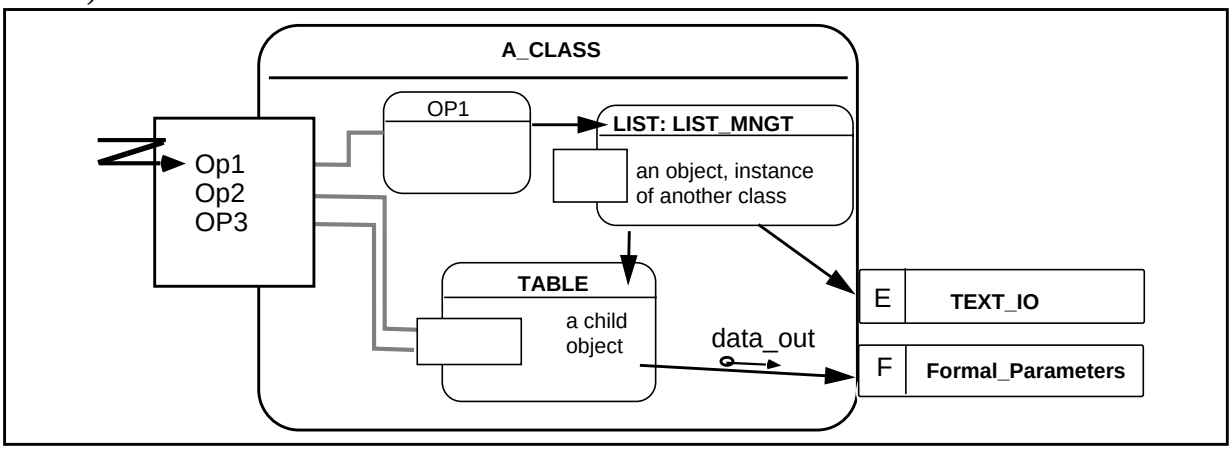


Figure 6 - A class as root object using environment and formal parameters object

1.1.3.6 Virtual Node Objects

The Virtual Node (VN) concept is the extension of the object concept (meaning a software module) to the system level (i.e. an executable, a process, a sub-system, a system...). The VN defines an *indivisible partition of the memory space and software units for distribution.*

Links between objects map procedures calls, even though links between VN map communication protocol. Communication between VN is only through protocol constrained operations. Some projects have used the VN concept to design a system where software, hardware and human activities have to contribute. In the same way than objects, a man or a hardware may provide services in response to requests.

A VN hierarchy (or Virtual Node Tree, VNT) gives the decomposition of a system into elements providing services and which constitute the whole system. Thus a parent VN may be decomposed into child VNs. At the lower level, a terminal virtual node consists in a system element which may either implemented by software, hardware or human actions. In case of software, the HOOD method allows to present the set of objects which are necessary to build that VN.
Thus, a terminal virtual node implemented by software is decomposed into those objects. We say that those **objects are allocated to the terminal VN**. This is not an include relationship but an allocation relationship. We can thus say that a terminal virtual node may map an executable (a UNIX process for example) built after compilation and link of all allocated objects.

Finally several VNs can be allocated to a physical processor, but one terminal VN cannot be distributed across several ones.

1.1.4 TEXTUAL FORMALIM

Each HOOD object is described in an associated Object Description Skeleton (ODS) which structures its properties into both informal and formal descriptions. Formal notations can be checked against informal descriptions and by tools.
An overview of the ODS is given in Figure 7 - below.

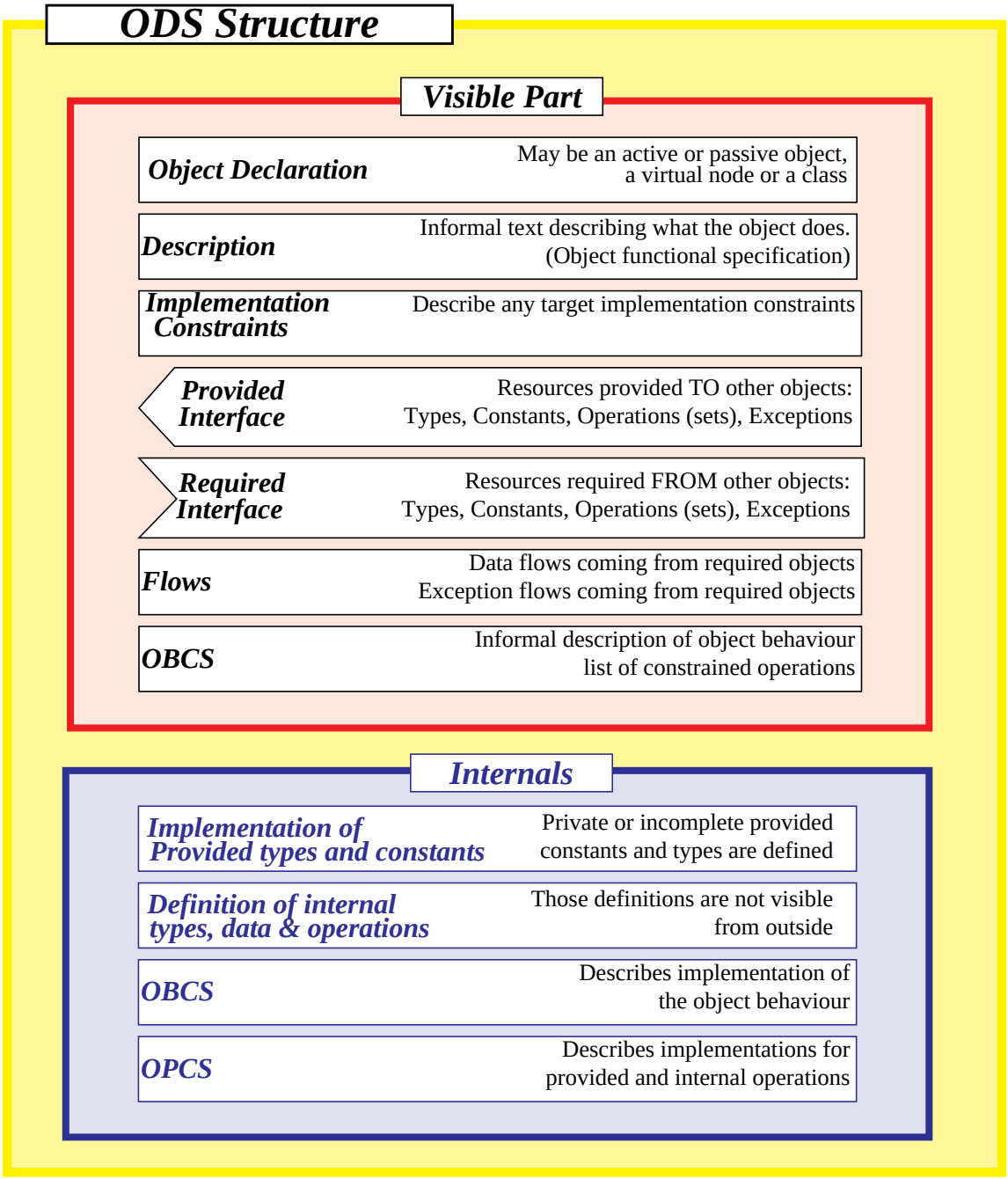


Figure 7 - Structure of a HOOD Object

OBJECT <Object_Name> is <Object_Type>
PARAMETERS -- Only for Class instances
TYPES, OPERATIONS, CONSTANTS

DESCRIPTION
-- Text in natural language giving all information for understanding and maintaining the object
IMPLEMENTATION_CONSTRAINTS
Natural language text giving constraints (memory limits, cpu, performances) for the object.

PROVIDED INTERFACE
-- Set of element provided to client objects. They are followed by a description in natural language
TYPES -- HOOD identifier,
CONSTANTS -- HOOD identifier,
OPERATION_SETS -- HOOD identifier

OPERATIONS --signature[1] in Ada,
EXCEPTIONS -- HOOD identifier

OBCS **-- Only for active objects**
DESCRIPTION
-- Text in natural language describing the behaviour of the object
CONSTRAINED OPERATIONS -- Sub set of provided operations which are constrained

DATA_FLOWS
--Textual Description of Data labels along the use relationship
EXCEPTION_FLOWS
--Textual Description of Exceptions labels along the use relationship

REQUIRED INTERFACE
For each Required Object, the subset of elements provided by the server object and really used in the current object
TYPES, CONSTANTS, OPERATIONS, EXCEPTIONS
INTERNALS

OBJECTS[2] **-- List all child objects implementing the object**

TYPES[3]**, CONSTANTS**[6]**, DATA**[4]**, EXCEPTIONS**[6]
--definition in target language of both provided elements (if not defined in the visible part) and internal one

OPERATIONS[5] **-- list of operation (with their signature in Ada) internal to the object**

OBCS **--Only for active objects**

PSEUDO_CODE[6] --with suitable notation
CODE -- in target language

OPERATION CONTROL STRUCTURES --Only for terminal objects

For each **OPCS**[7]
Description
Used_Operations **-- Operations (internal or external) used by that operation**
Exceptions_Propagated
Exceptions_Handled
PSEUDO_CODE -- in PDL
CODE -- in target language
END <Object_Name>

1. Signature = syntactic specification of an operation. By default it is expressed in Ada syntax, what ever target language
2. This field is empty for terminal objects
3. These fields provide for definition of implementation of associated structures in terminal objects, and for definition of 'implemented_by' relationship for non terminal ones.
4. This field only exists for terminal objects.
5. This field declares the “implemented-by” links of parent operations down to child ones. For terminal objects additional internal operations may be declared here in the process of step-wise refinement of provided operation control structures.
6. The notation can be a graphical one or textual one, allowing to specify formally control (Ada, Petri nets, Finite state Automata, Esterel, Temporal Logic) and provide for automatic generation of code.(see *[2]* for more details).
7. These subfields are filled for each operation internal or provided ones.

Figure 8 - ODS Outline

The different descriptions fields of the ODS allow to capture and refine object properties, and can be processed by dedicated toolsets in order to:

- generate target code units automatically
- generate various documents for verification (cross reference tables, dictionaries, indexes)

1.1.5 AN ODS ILLUSTRATION- THE STACK OBJECT

In the following we give an illustration of an ODS associated to a terminal object called STACK, modelling a STACK abstract data type and which is represented as Figure 9 -, with two operations PUSH and POP constrained by the object's internal state.

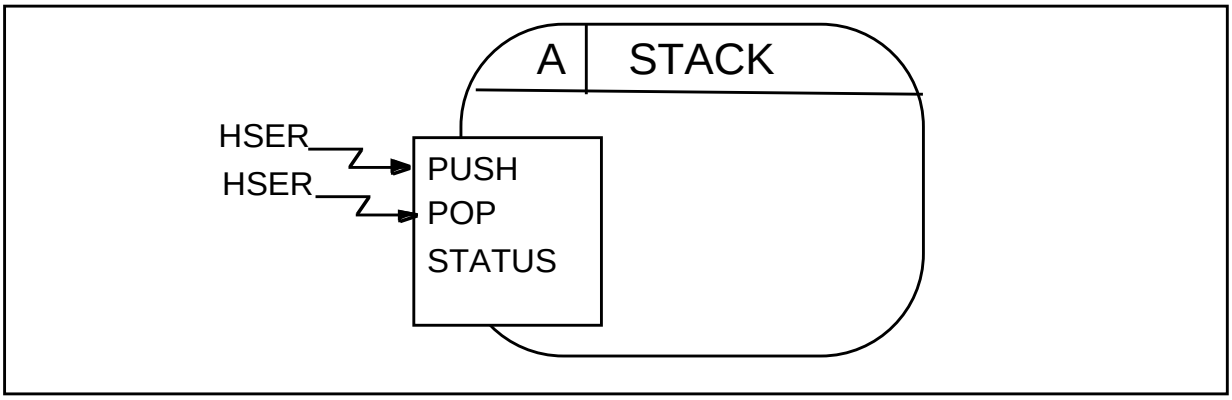


Figure 9 - Graphical representation for the active object Stack

The STACK object behaviour can be modelled by a state transition diagram as in Figure 10 -, where transitions are exclusively associated to operation execution.

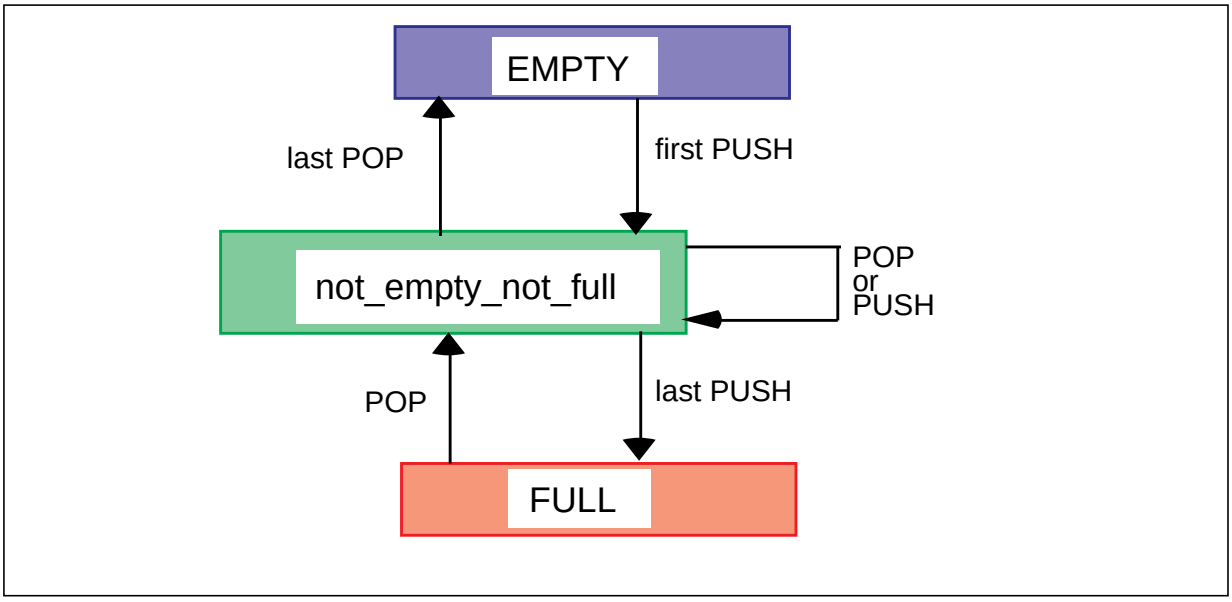


Figure 10 - Object behaviour for STACK

The OBCS implementation can be reduced in this case into one Ada task implementing the state transition diagram and the associated Ada code could be described in the OBCS CODE field(see Figure 11 -).

The ODS of Figure 11 - shows the declarations of OBCS, and declaration of provided and required types of object STACK.

```
OBJECT STACK IS ACTIVE
DESCRIPTION
   -- abstract data type STACK with no encapsulation of data instances
IMPLEMENTATION_CONSTRAINTS
   --The target system shall be an Ada system with full Ada83 tasking
PROVIDED_INTERFACE
   TYPES
      T_Status is (BUSY, IDLE, UNDEFINED);
      -- definition of provided type to hold the status of the stack code.
      type T_DATA is array (integer <>) of ADT_Element.T_Element;
         --data structure to hold values of the type
      type T_STACK(MAX: integer) is --type STACK implementation
      record
         DATA: T_DATA (1 .. MAX); -- only needed space is declared
         TOP: integer:=0;
         SIZE: integer:=MAX;
      ETAT: PROJECT_ENV.T_STATES;
      end record;
      -- a client declares stack objects as
      --Client_Stack: T_STACK(150);
   OPERATIONS
      PUSH (STACK: in out T_STACK; Element: in T_Element);
      POP(STACK: in out T_STACK; Element: out T_Element);
      STATUS(STACK: in out T_STACK) return T_Status;
      EXCEPTIONS
      X_FULL raised by PUSH when more then MAX elements in the STACK
      X_EMPTY raised by POP when no elements in the STACK
   OBJECT CONTROL STRUCTURE -- visible part of OBCS
      DESCRIPTION
      PUSH and POP operations should only be activated when the state of STACK is non(EMPTY) for POP or n(FULL)
   for PUSH
      CONSTRAINED_OPERATIONS
      PUSH is constrained by HSER. -- HSER is additional to state constraints
      POP is constrained by HSER- -- because the access must be protected in a multi-client/ multi thread context
-------------------------------------------------------------------------------------------------------------------------------
-- END OF STACK USER MANUAL
-------------------------------------------------------------------------------------------------------------------------------
REQUIRED_INTERFACE
   OBJECT ADT_ELEMENT
      TYPES
      T_Element;--l STACK requires type T_Element, provided by object ADT_ELEMENT.
   OBJECT ADT_SEM
      TYPES
      T_SEM;--STACK requires type T__SEM, provided by object   ADT_SEM.
      OPERATIONS
      P,V;
   OBJECT PROJECT_ENV
      TYPES
      T_States;--STACK requires type T_STATES, provided by object PROJECT_ENV stores basic type definitions for
   this project ( alittle more than the Ada standard package)

INTERNALS
   -- not shown here, see below
```

Figure 11 - ODS for STACK (User Manual)

The INTERNALS part of the STACK ODS in Figure 12 - shows the declaration of internal types, data and OBCS code.

```
INTERNALS -- hidden part of the object:
    OBJECTS None; --NO CHILD OBJECTS
    TYPES None; -- no internal types
    CONSTANTS None; -- no internal constants
    DATA
        GARDE: ADT_SEM.T_SEM:=FREE; --exclusion semaphore
        STATUS: T_Status; -- class variable

    OPERATIONS None; -- no internal operations
    OBCS
        PSEUDO_CODE
        --see OBCS STD in figure A2.2
        CODE --of OBCS --
        task body OBCS is
        begin
            loop --
                select -- choice according to current state see Figure 10 - above
                    when Not_EMPTY | FULL =>
                    accept POP (One_Element: out T_Element) do
                        OPCS_POP (One_Element);--associated OPCS body
                    end POP;
                or when not(FULL) =>
                    accept PUSH (One_Element: in T_Element) do
                        OPCS_PUSH (One_Element);-- associated OPCS body
                    end PUSH;
                end select
            end loop;
        end OBCS;

OPERATION CONTROL STRUCTURES
    OPERATION STATUS return T_Status is --OPCS of POP
        DESCRIPTION
        Get status of class STACK code
        USED_OPERATIONS None;-
        PROPAGATED_EXCEPTIONS -- None;-
        PSEUDO_CODE None; - --see code;
        CODE-- in Ada
            begin
            return STATUS;
        END --of opcs STATUS
-- see other OPCS on next page
```

Figure 12 - ODS for STACK (Internals)

```
   OPERATION POP is --OPCS of POP
      DESCRIPTION
      Remove an Element from the data structure T_DATA
      USED_OPERATIONS
      None;--
      PROPAGATED_EXCEPTIONS
      X_STACK_UNDEFINED;
      PSEUDO_CODE--
          if [the STACK is not EMPTY] then
              [put STACK in BUSY state]
              [remove Element from STACK_DATA]
              [put STACK in IDLE state]
          else
              [put STACK in UNDEFINED state]
          end if;
      CODE-- OPCS_body is
          begin
          if STACK.TOP>0 then
              STATUS:=busy; -- STACK code is BUSY
              Element:=STACK_DATA(STACK.TOP);
              STACK.TOP:=STACK.TOP-1;
              STATUS:=idle;-- STACK code is IDLE
          else
              STATUS:=undefined; --STACK code UNDEFINED
              raise X_EMPTY; --impossible case
          end if;
      END --of opcs POP

   OPERATION PUSH is --OPCS of operation PUSH
      DESCRIPTION
      Put an element onto STACK
      USED_OPERATIONS None;--
      PROPAGATED_EXCEPTIONS
          X_STACK_UNDEFINED;--
      PSEUDO_CODE--
          if [the STACK is not FULL] then
              [put STACK in BUSY state]
              [PUT Element in STACK]
              [put STACK in IDLE state]
          else
              [put STACK in UNDFINED state]
          end if;
      CODE -- -- OPCS_body
          begin
          if STACK.TOP<STACK.SIZE_STACK then
              STATUS:=busy; --class code in BUSY state;
              STACK.TOP:=STACK.TOP+1;
              STACK_DATA(STACK.TOP):=Element
              STATUS:=idle;--class code in IDLE state
          else
              STATUS:=undefined; -class code in UNDEFINED state
              raise X_FULL; --impossible case
          end if;
      END --of opcs POP
END_OBJECT STACK;
```

Figure 13 - ODS for STACKs (Internals continued)

1.1.6 FROM ARCHITECTURE TO TARGET IMPLEMENTATION

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

- HOOD entities declaration such as OBJECTS, PROVIDED REQUIRED INTERFACE (TYPES, OPERATIONS and EXCEPTIONS) are “target language independent” and can be used to build a target unit architecture using visibility and encapsulation mechanisms as available in the target language/system.
- The associated body fields are “target language dependent” and are directly included in the right place within the target skeleton architecture.
  - *mapping HOOD objects to target units.* Encapsulation mechanisms provided by the language and target environment are used. For e.g. a Ada package, or a C or C++ module(defined by two files with extension.h and.c) is associated to each HOOD object.
  - *implementation of object relationships.* INCLUDE relationships will be implemented by encapsulation mechanisms, visibility mechanisms such as the use of Ada “with” clauses, and possibly direct file inclusion at source code level. USE relationships will be implemented trough target language mechanisms controlling intra and inter unit visibility, and possibly by using code inclusion. (as e.g. in C or C++).

HOOD has standard code generation rules for the Ada language, since this latter comprises a tasking model, thus making the generated code as an executable model on Ada targets, but also a specification model for alternative implementations on other targets. Specific rules for target systems other than Ada are described in Section 2.12.

OBJECTS	Ada Units
OBJECTS passive	**package**
OBJECTS active	**package**
OBJECTS active (terminal)	package which includes one or more task unit
Operation (non constraint)	**procedure** or **function**
Operation (constraint)	**task entry**
Exception	**exception**
CLASS OBJECTADT	(package)or **generic package**
CLASS OBJECT INSTANCES	package instantiation
RELATIONSHIPS	
USE	**with** clauses at the lower level of visibility
INCLUDE	nested or withed packages

Figure 14 - Mapping between HOOD and Ada entities

1.1.6.1 General Implementation Rules

The code generation rules differ for parent or for terminal objects:

- *for parent objects***:** the associated code is generally a specification unit mapping the provided interface of the object. These specifications are implemented using *renames* Ada clauses[3] onto specification units associated to child terminal objects as illustrated in Figure 16 -.
- *for child objects:* by default are generated as “flat” Ada packages not included in parent packages. As a result parent packages “**with**” those associated to their children as illustrated in

[3]except for **type** renaming which is not supported in Ada Parent types may be implemented as **subtypes** and **attribute access functions** to terminal child types. But a standard solution is to make parent unit have visibility on terminal child types through with clauses. See also recommendations of HADA]

Figure 15 - and Figure 16 -.

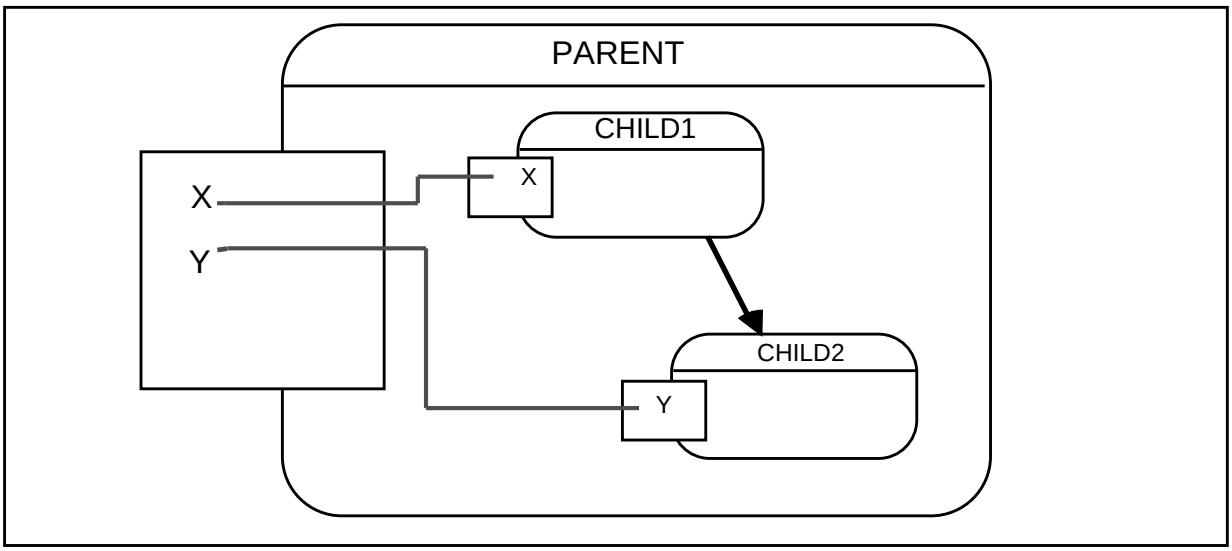


Figure 15 - Code generation principle for Parent object

```
with CHILD1; --visibility on CHILD1 generated as a library unit
with CHILD2; --visibility on CHILD2 generated as a library unit
package PARENT is
    -- specification unit associated to object PARENT
    procedure X renames CHILD1.X;
    procedure Y renames CHILD2.Y;
end Parent;
```

Figure 16 - Code generation principle for Parent object (standard generation)

*An alternative solution is to nest the child packages into the body of the parent package***,** whose specifications are then implemented by extra code mapping child ones, what might become rather inefficient as illustrated in Figure 17 -.

```
package body PARENT is -body associated to object PARENT
    package CHILD1 is --specification unit associated to object CHILD1
        procedure X;
        • • •
    end CHILD1
    package CHILD2 is --specification unit associated to object CHILD2
        procedure Y;
        • • •
    end CHILD2;
    procedure X is
    begin
        CHILD1.X; -- call to CHILD operation
    end X;
    • • • •
end PARENT;
```

Figure 17 - Code generation principle for Parent object (Nesting Child packages in parent Body))

This solution has however to be used for children associated to generic packages in order to give visibility to the formal parameters (parameters of generic Ada units).

1.1.6.2 Implementation of Constrained Operations

When an operation execution is constrained, its activation is related to the object state and/or to a given communication protocol to be enforced between the current object and the client. The implementation principle, illustrated in Figure 18 -, is to introduce additional code upon the body of an OPCS, which will handle these constraints as the body of the associated operation will execute.

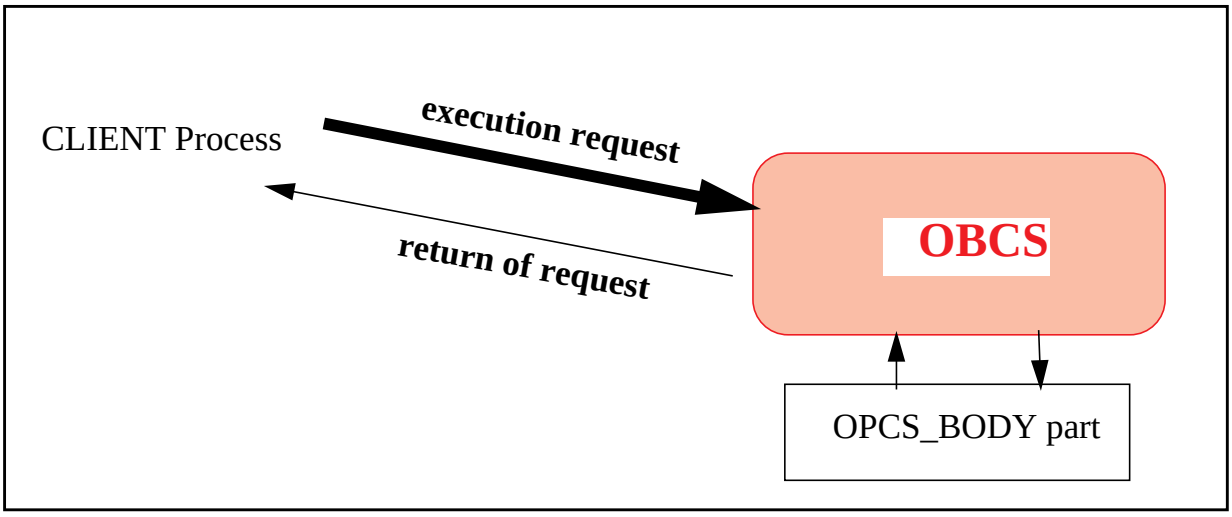


Figure 18 - HOOD 3 Code structure for constraint operations

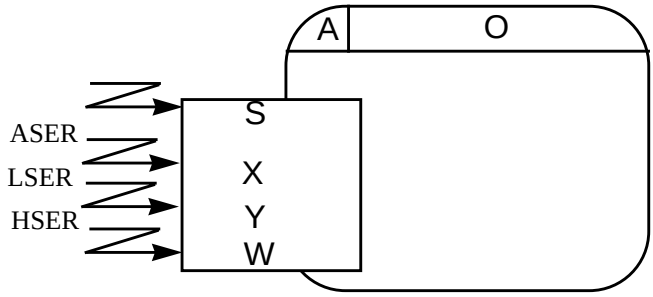
The idea is that most of **the code associated to the constraints could be automatically generated and can be defined once for all**, so that the designer and coder has only to do with the core OPCS body.

Moreover, this separation allows to do feasibility analysis, prototyping of real-time behaviors completely in parallel with the development of the functional code of a system (the pure opcs bodies)

The implementation schema is that the OBCS is implemented as a (or more) target unit (in Ada it is a package or task generally named OBCS) which handles execution requests and operation execution via calls to an associated procedure named by convention OPCS_<Operation_Name>.

Figure 19 - below illustrates these principles for an active object O with four constrained operations S, X,Y and W.

By convention, there is one such OBCS per object, but in case of resource shortage, its is always possible to group several OBCS into one. HOOD standard generation rules*[1]* define the OBCS as an Ada package, possibly reduced to one task unit, and the OPCS header code as a rename of an OBCS entry hiding the OBCS to clients, whereas the OPCS body code is defined in a procedure called OPCS_OpName. The standard scheme may be quite resource consuming especially when guards are used to handle state constraints; accordingly a schema based on FSM (Finite State Machine) has been defined and detailed in *Appendix A3.2.1*.




```
package O is
      task OBCS is
          entry S;        -- state constrained operation
          entry X;        -- ASER constrained operation
          entry Y;        -- LSER constrained operation
          entry W;        -- HSER constrained operation
      end OBCS;
   procedure S renames OBCS.S;
   procedure X renames OBCS.X;
   procedure Y renames OBCS.Y;
   procedure W renames OBCS.W;
end O;

   package body O is
      • • •
      procedure OPCS_S is separate;
      procedure OPCS_X is separate;
      procedure OPCS_Y is separate;
      procedure OPCS_W is separate;
      task body OBCS is
      begin
            loop
               select
                  accept S;--State constraint
                     OPCS_S; --return with operation executed if State was OK
                              -- state can also be handled as guards applied to the entry
               or
                  accept X;-- State constraint and ASER constraint
                              --return without execution if State is not OK
                              -- state can also be handled as guards applied to the entry
                     OPCS_X; --ASER is roughly implemented in Ada
               or
                  accept Y do; --LSER constraint
                     -- save parameters
                  end Y; -- release client process
                  OPCS_Y;
               or
                  accept W do --State constraint and HSER constraint
                              --return without execution if State is not OK
                              -- state can also be handled as guards applied to the entry
                     OPCS_W;
                  end W; --release client process
               end select;
            end loop;
      end OBCS;
   end O;
```

Figure 19 - Code generation principles for terminal active object

1.2 THE HOOD DESIGN MODEL

The HOOD method gives a framework for the project design organisation. Principles of that organisation are developed in that section. They will be the basis for the HOOD design process presentation.

1.2.1 SYSTEM TO DESIGN

Before going into the HOOD design process, we first introduce the System To Design notion (STD). This is the System (in the large sense) which has to be designed from a given specification. It may be a part of the whole project.
A HOOD **System To Design** (STD) may be first seen as an object defined by its interface with its environment. The environment is considered as an other part of the project, not designed for the purpose of the current system, and for which the root is seen as an environment object from the STD. The object is then decomposed into child objects, which can themselves be further decomposed (until the specifications of child objects can be directly implemented in the target language). A STD can thus be represented as a root object of a HOOD *design tree* (HDT) where branches represent parent objects decomposed into child objects, and where leaves are terminal objects which are no more decomposed.
Figure 20 - hereafter represents a STD as a root of a hierarchy of objects within a whole project including other hierarchies (other HDTs).

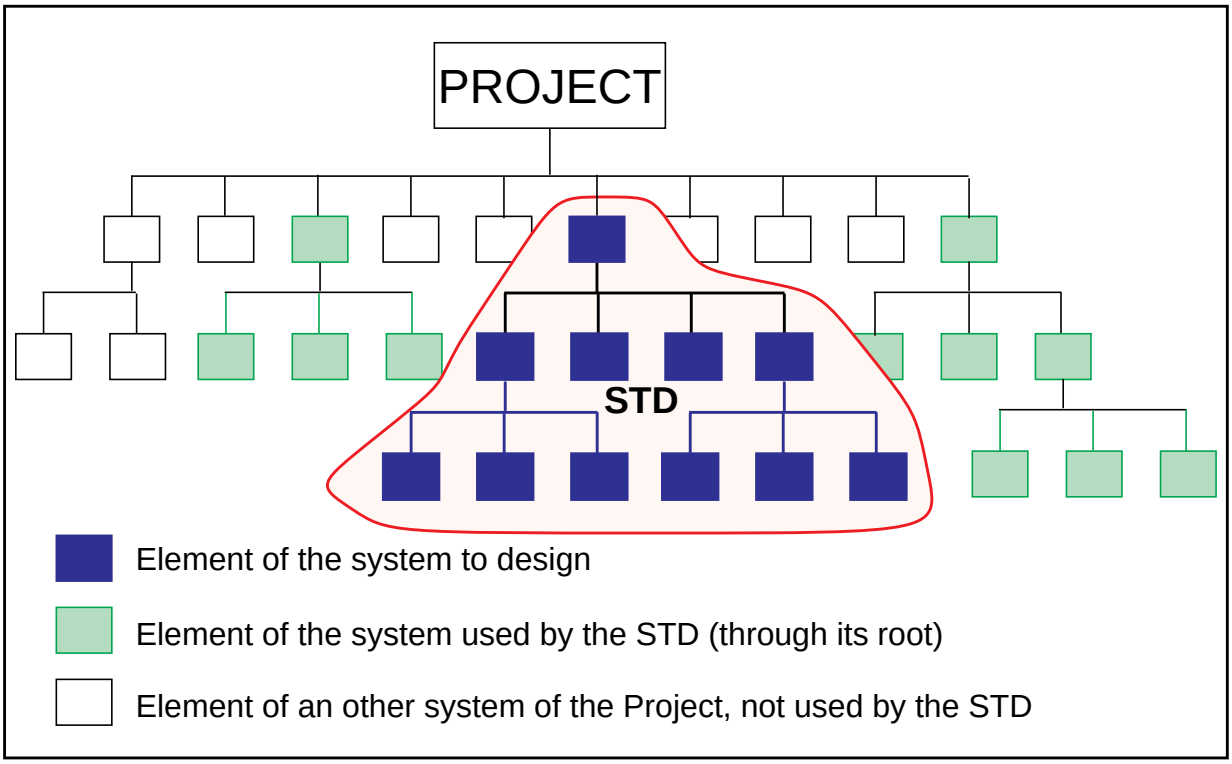


Figure 20 - A System To Design within a whole project (Client/server use relationships are not shown)

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

1.2.2 SYSTEM CONFIGURATION

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

Classes[4] define templates of objects. Classes are not objects and cannot be used. Thus, the class hierarchies (Class Design Tree or CDT) cannot be defined within the object space. The CDT are described in an other space: the class *object space*.

Following similar description principles, the Virtual Node hierarchies (VNT) are defined in a separated *Virtual Node space* allowing to define distributable entities, memory partitions *or* heavyweight processes as defined in [3].

Finally, VNs have to be allocated to physical nodes of the physical architecture. This architecture is defined as the *Physical space*.

A whole project organisation may thus be defined through the HOOD design model (see Figure 21 -) as:

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

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

[4]. Classes in HOOD4 are termed GENERICS, thus avoiding a clash with the OO terminology

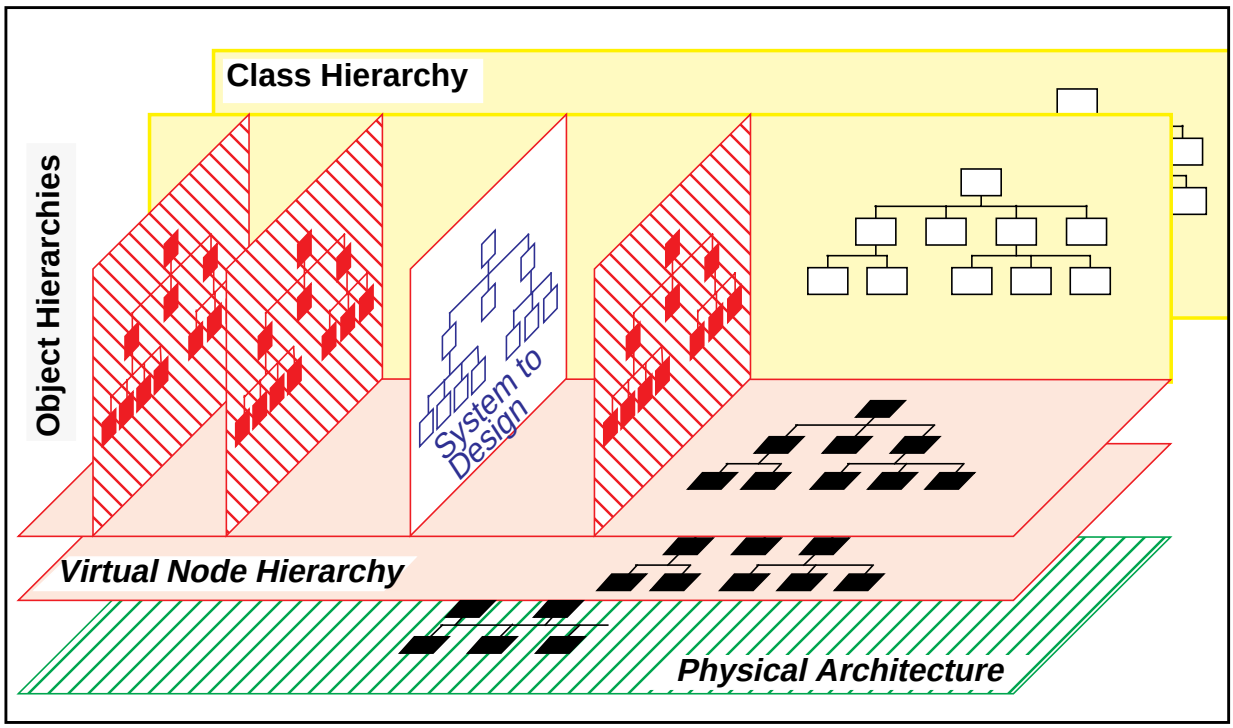


Figure 21 - The HOOD design model into a set of spaces and hierarchies.

1.3 THE HOOD DESIGN PROCESS

The HOOD method proposes a process to guide the designer in his job. This process allows to get a good design what ever the size and the complexity of the project. This approach orients to the best choices. Two different processes are imbricated:

- **a general process driving the development approach, called "Overall HOOD Design Process"** in the following**.** This process describes the approach and activities to perform along the architectural design phase in order to organise the system according to the development constraints of the project (sub-contracting for example). It allows to define several models of the system down to the system configuration. This approach is more detailed in Section 1.3.2 below,
- **the basic HOOD Design Process** which defines the activities to decompose a given object into children, from a top object to all terminal objects, and to get thus a HOOD Design Tree (HDT or CDT). This process is applicable for all objects and classes and is more detailed in Section 1.3.1 below.

1.3.1 THE BASIC HOOD DESIGN PROCESS

The basic HOOD design process consists in building a HOOD Design Tree (HDT or CDT). It is globally top-down. The system to design is first defined as a high-level object called *root object*, and then is broken down into several lower level objects up to they can be directly implemented by target language units and environment services.
As seen in Section 1.2, the system to design can be represented by a *HOOD design tree* where branches represent parent objects broken in children and leaves represent terminal objects, which are no more decomposed.
The process of decomposing one object into child objects is called the **basic design step**. Thus, it is necessary to perform a succession of basic design steps to build the whole HDT.

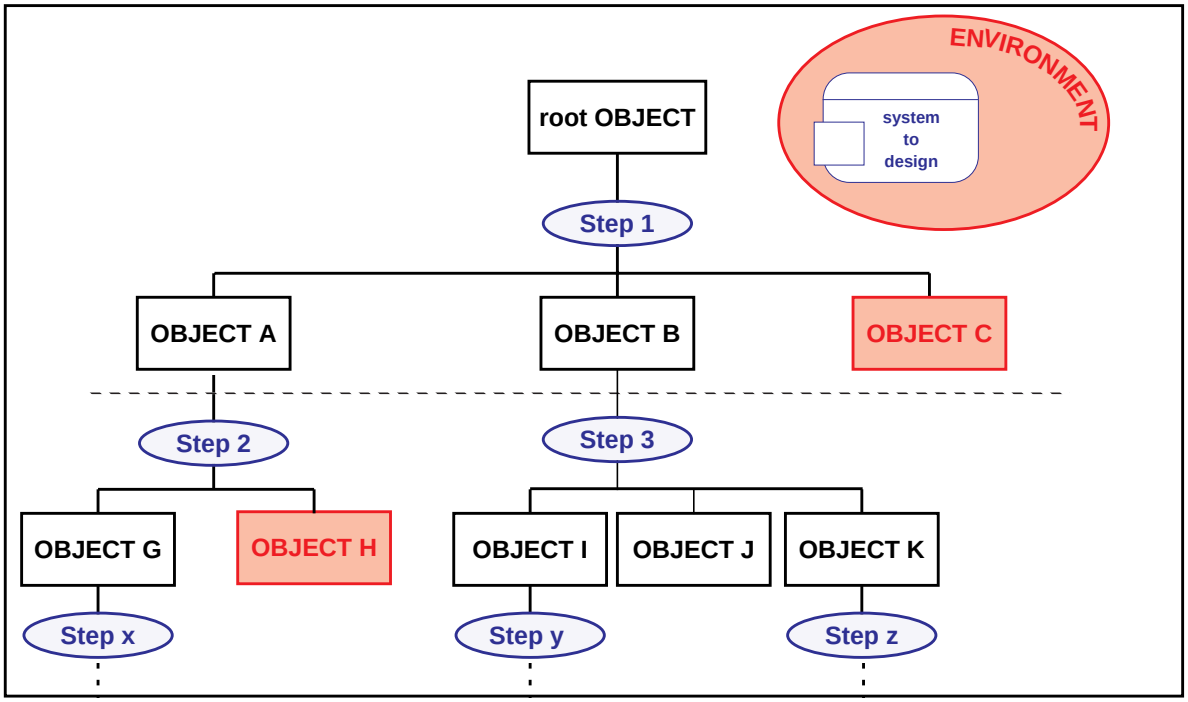


Figure 22 - HOOD Design Tree as decomposed from the ROOT object

1.3.1.1 The Basic Design Step

Originally inspired from OOD techniques *[BERARD85]*, *[BOO86]*, *[BERARD85]*. The basic design step allows to decompose a well defined object (in term of provided and required services, behaviour and functions) into children. It comprises a set of elementary activities allowing to continuously transform an object from its definition up to its implementation. Some results of those activities (textual and/or graphical pieces) will later be assembled to setup ODS and design documents. The basic design step activities are structured according to life cycle phases.

The method discussed hereafter will not find a design solution, **but will help the designer to build the best one (if any) according to project constraints.**

The basic design step consists in a set of (initially) consecutive and different activities:

- **Activity 1:** Definition and Analysis of the Problem: This activity consists in well understanding the requirements before starting the design,
- **Activity 2:** Elaboration of an Informal Solution Strategy: This activity consists in describing how the system is working,
- **Activity 3:** Formalisation of the Strategy: This activity consists in organising (through a HOOD diagram) the design of the solution previously described,
- **Activity 4:** Solution Refinement: This activity consists in refining the previous solution taking into account particular project constraints,
- **Activity 5:** Formalisation of the Solution: When the previous refinement is finished, this activity consists in detailing the solution in terms of textual descriptions (ODS),
- **Activity 6:** Solution Analysis and Justification

Main ideas of that process are to quickly build a first solution (end of third activity) which is the basis for a refinement(activity 4). The method will constrain this solution according to good software engineering and design principles in order to achieve a better solution. The fourth activity then consists in modifying this solution according to other constraints in order to get the final solution.

Figure 23 - below gives a summary of activities of the basic design step and main outputs, whereas a detailed discussion is given below.

Outputs of activities are produced during the basic design step as text sections, but may not all be included in a relevant ADD. It is generally interesting to keep that information for the project history of for design justification (Activity 6).
In order to ease readability of the design documentation, each activity output is numbered. A possible numbering convention prefixes the activity with an H[5] :

- identify activity outputs as:
  - H1 for Problem Definition
  - H2 for Informal Solution Strategy
  - H3 for Formalisation of the Strategy
- identify subsections activities as H1.1 H1.2, H3.1, H3.2 etc

[5]. H for HOOD Design Step

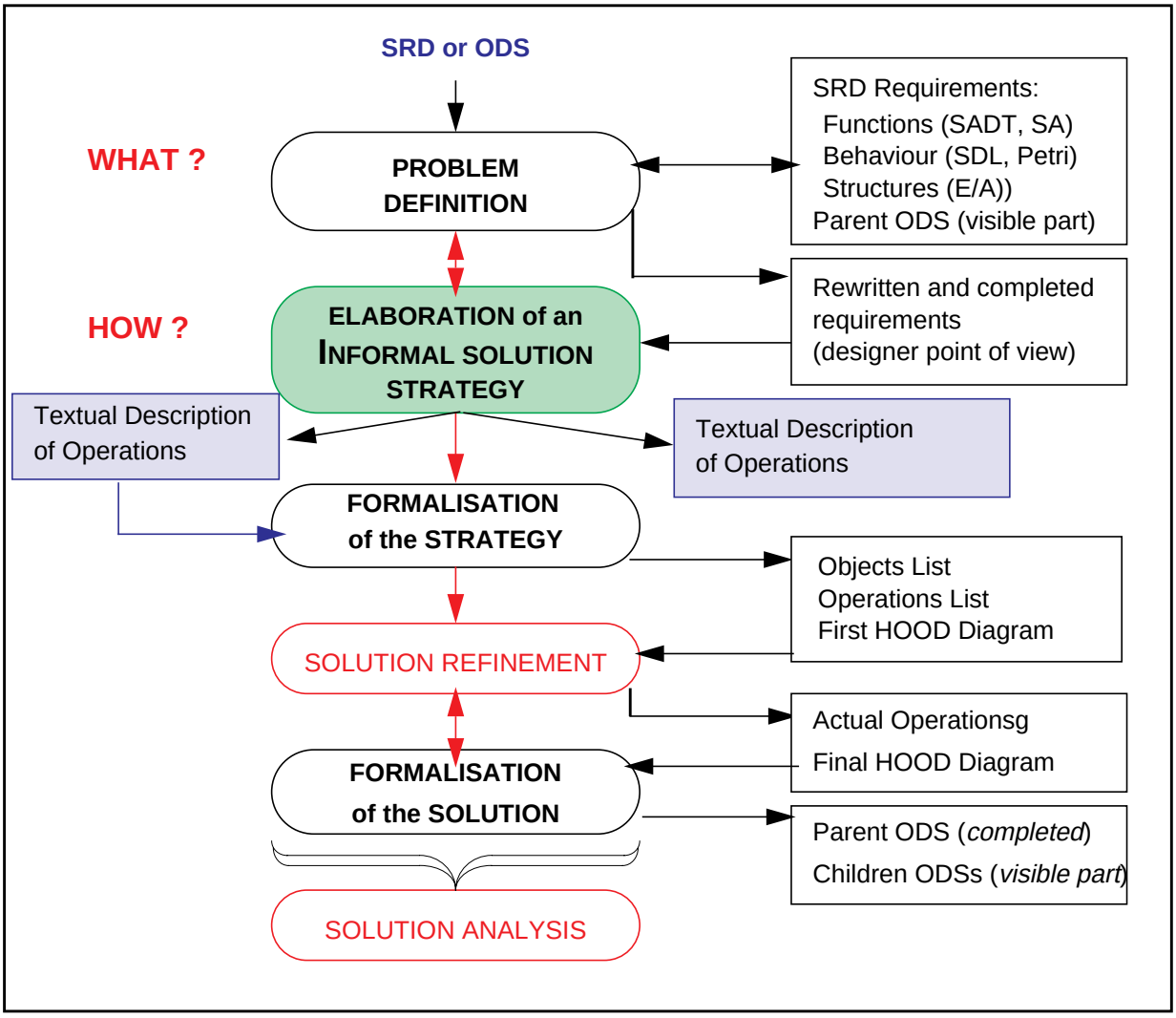


Figure 23 - The HOOD design activities and associated outputs

Activity 1: Problem Definition

Activity Inputs

A basic design step may start either after a specification phase (beginning of the design phase), or after a previous design step (during the design phase).
When starting from **a specification** (SRD), the designer has to understand and classify requirements (functional, behavioural, structural and interfaces) and the current activity is essential.
When starting during the design, from **a child ODS**, the designer has to understand information described within the parent ODS visible part (Object description, provided and required interfaces, OBCS description). Requirements are already reorganized and are at a lower more detailed level.

Activity

"Understand the problem to solve before jumping on a solution"
"Design Oriented Analysis"

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

- **Statement of the Problem** (SOP): The designer states the problem in one correct sentence giving a clear and precise definition of the problem as well as the context of the system to design.He must identify objectives of the current design (in terms of portability, reusability, etc...).
- **Analysis and Structuring of the Requirement Data**: The designer gathers, analyses and organises all the information relevant to his problem, *clarifying all points which are not yet clear*.When starting from high level requirements, his first task is to define the environment of the system-to-design (provided and required interfaces). Then, he organizes the requirements into *functional, behavioural and non-functional* ones (such as performances) and produces a synthesis. He performs *design sensitive analysis* upon them and possibly produces a user manual outline of the system-to-design.

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

Activity Outputs (H1)

- **Statement of the Problem** (H1.1):
  - Description of the problem and its context in a few sentences.
  - List of main design objectives.
- **Analysis and Structuring of the Requirement Data** (H1.2):
  - analysis and definition of interfaces
  - HOOD context diagram-
  - analysis of functional constraints[6]
  - analysis of behavioural constraints[7]
  - analysis of structural constraints[8]
  - analysis of non functional constraints[9]
  - user manual outline

Only the System to design environment and the statement of the problem may have to be included in a relevant ADD.

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

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

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

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

Activity 2: Elaboration of an Informal Solution Strategy (ISS)

Activity Inputs

At this stage, all necessary elements are well organized (either in a note coming from the previous phase, or in the child ODS coming from the previous design step) and the design activity may really start.

Activity

"Outline a solution, put it down as a text and work it out"

This phase has as goals the expression of *a solution* that the designer should have started to imagine (to *phantasm* about) from the earlier phases. The designer shall identify the main abstractions and actions and give a scenario of solution accordingly.
Such a strategy shall be expressed in natural language in order to explicit clearly **how the solution will work**, without requiring any organisation (don't speak about objects in a first issue). The purpose is to explain what happens when the current object services are required in terms of data and actions on those data.

Good example: *On IT reception from the Start push button, the EMS inits bargraphs, creates and starts a timer which triggers the monitoring each second.*
Bad example[10]: *The EMS is composed of a Sensor object which encapsulate all actions on the physical sensors.*

This activity will not allow to build the final design but a first issue of that design. The designer has mainly to consider the quality of the design more than its efficiency (this will be the purpose of the next activities). Design quality relies on ISS text (actions and data).

Activity Outputs (H2)

A clear and concise text expliciting the solution. This text will have to evolve as the design matures during the design step as well as to be consistent with the graphical description elaborated during the third activity.

[10]. The structure of the system is described; we still do not know how it works!

Activity 3: Formalisation of the strategy

Activity Inputs

Input of the current activity is an informal textual description of **a solution scenario**.

Activity

"Refine and work out your solution"

This phase has as goals the extraction of the major concepts of a solution strategy in order to come smoothly to a formalised description of the previous solution through a HOOD diagram. The idea is that a solution which can be expressed clearly in natural language is an almost mastered solution.

A good set up of the previous activity leads into identification of objects through nouns and operations provided by those objects though verbs. Thus, the current activity may consist in:

- **identifying nouns** (ex: «The user requires a chocolate from the vending machine»):
  - nouns which provide services are generally good candidates for objects (ex: «vending machine» provides services),
  - nouns which does not provide services define main dataflows of the design (ex: «chocolate» does not provide service. It is a data coming from the vending machine to the user).
- **identifying verbs**: All verbs are operations. Those operations use and/or return data (Ex: «requires» verb allow to define the require operation. It returns «chocolate» data).
- **grouping operations and verbs**: This task, started from the two previous lists of objects and operations gives a structured representation of the strategy and helps completing a graphical description of the solution (ex: The «require» operation is provided by the «vending machine» object).In certain cases, it is difficult to join operations to objects and it could be interesting to add objects (such as controller, monitor, scheduler) to encapsulate those operations.
- **drawing graphical representation of the design**. This task consists into drawing, (within the parent object), objects identified during this activity with their operations, using the HOOD graphical formalism. *Use* links between child objects (and potentially uncles or environment objects) and *implemented_by* links between parent operations and children operations are identified. Dataflow (or exception flows) along the use links should also be identified according to the understanding of the activities/functions allocated to the objects[11].

Activity Outputs (H3)

The produced graphical HOOD diagram must be consistent with the ISS text (H2).

[11]. This understanding can be gained if the ISS text was well written or is well understood.

Activity 4: Refinement of the solution

Activity Inputs

Current solution of the design expressed with consistent textual description of the ISS (H2) and graphical description (H3).

Activity

"Review and agree on your solution, before formalizing it!"

Output of the previous activity may not be considered as the final HOOD design but as a preliminary version used to support discussion between designers. Indeed, designers have to introduce specific project constraints, behavioural and dynamic ones (attributes relative to parallelism, synchronism, periodic execution), or even build prototypes to check particular points and then to modify the solution.
Thus, this activity has to be included between the formalisation of the first solution and the finalization of the last one. Advantage of the previous activities are that all designers work from a same design (graphical and textual) which has a good quality level.

It is relevant to stress that the graphical description goes with the textual descriptions (and vice-versa) and that the consistency between these two kind of representations shall always be ensured.

This activity also consists into review and author/reader cycles on produced HOOD diagram and textual descriptions.

An important part of that activity consists in expressing reasons of design decisions (scenario and organisation) which may be questionable or is not obvious. This allows to justify the design choices and to achieve reasons of those choices.

Activity Outputs

Outputs of this activity are a consistent:

- textual description of the solution scenario,
- graphical description of the solution organisation,
- justification of design decisions.

Activity 5: Formalisation of the solution

Activity Inputs

Once the formalisation of the strategy has been completed, the solution is well defined through the textual ISS, the objects, provided and required interfaces identified through the HOOD diagram, the designer can formally capture his solution.

Activity

"Detail the solution through ODS"

The goal of this phase is to achieve a description of the solution with the detailed characteristics of the objects being formalised in the ODS fields. The capture of this formal description consists in filling each field of the **Object Description Skeleton** (see Section 1.1.4 above). In order to have knowledge of the environment of the ODS to be filled, it is important to fill ODS from bottom (objects which does not use other objects) to top (objects which are not used).
The designer should:

- describe the object in terms of «what the object does»,
- define constraints operations and describe the OBCS (if any) explaining the synchronisation and relationships between constrained operations,
- describe the provided operations (and Operation_set) «what the operation does and error cases»,
- refine the subset of operations really required (required interfaces),
- define parameters of provided operations and deduce provided types and exceptions,
- define and describe provided types and deduce potential provided constants,
- describe provided constants and exceptions.

Informal comments may be added to describe the semantic behaviour, to provide useful information for further implementation or to justify possibly implementation decisions.

Activity Outputs (H4)

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

At the end of this phase, the ODS of the parent object is fully and formally described (ODS internals part is completed) and child ODSs are preliminary filled (ODS visible part is completed).

Activity 6: Analysis of the Solution

Activity Inputs

Any outputs of the previous phases.

Activity

"***Critically Review the solution"***

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

- Design Justification,
- Re-structure the solution for a best design quality: minimize top level objects (not used), maximize bottom objects (highly used), maximize consistency and minimize coupling, avoid cycles, etc...
- Consistency and Completeness Validation,
- Identification of Re-usable Objects,
- Identification of Potentially Generic Objects,
- Post-Analysis Design Update possible updates in design step M and M-1
- Traceability entries: this is the right time to define which requirement the current design is fulfilling. The designer can thus define entries in a traceability matrix or directly within the ODS,
- Risk analysis in order to identify critical issues in the solution in terms of technical and management risks. For technical risks concerning failure management, detection means and recovery actions have to be studied and the solution have eventually to be updated.

Activity Outputs

Associated texts may be put in the DESCRIPTION ODS fields as informal text.

Activity 7: Continue the decomposition

For each identified child object, analyse if it has to be decomposed and then restart the activity 1. In the other cases, see Section 1.3.1.3. below.

1.3.1.2 The Basic design step applied to a root object

Unlike the other objects, the root object (STD) must be defined prior to be decomposed. Thus, the «Problem Definition» activity needs to define the interfaces (provided and required) of the root object. The designer puts the emphasis on the environment of the System To Design (STD); he has to produce a diagram showing the STD within its environment.

1.3.1.3 The Basic design step applied to terminal object

The basic design step applied to the terminal objects is shorter and consists in detailing the internals with declarations, internal operations and pseudo-code. Each basic design step activity is restricted as follows:

Activity 1: Problem Definition

The designer should gather all the information related to the terminal object.

Activity 2: Elaboration of an Informal Strategy

The purpose of this activity is to provide a short description for the internals OBCS and OPCS(s) of the object, derived from their description in the ODS visible part. Those description are implementation oriented. The designer has also to give internal declaration (not defined in the ODS visible part).
The goal defined in the general scheme of creating a solution strategy is not applicable here, as the terminal object is not decomposed any more.

Activity 3: Formalisation of the strategy

This activity consists in giving, for each OPCS (corresponding to provided operations) and OBCS, a detailed description of implementation with a pseudo-code. This description highlights control structures.

Activity 4: Refinement of the Solution

From the previous OPCS description, the designer has to identify new internal types, constants, exceptions, and new internal operations in order to complete the ODS internals description.

Activity 5: Formalisation of the Solution

This activity consists in finalize description and definition of internal declaration (types, constants, exceptions and data), description and pseudo-code of OBCS and OPCSs.

Activity 6: Analysis of the Solution

Applied to the internals of the ODS, this activity consists in a author/reader cycle on Control Structures algorithms. Analysis will also highlights introduction of internals declarations (types, constants, exceptions data) and internal operations.

1.3.1.4 The Basic design step applied to the other types of object

The way the HOOD basic design step is applied to other types of objects may slightly vary:

- ***Op_control:*** An Op_control is a terminal object which is used to map parent operation to multiple child operations. Activity 2 to 4 are relevant for such an object. In Activity 2 “Elaboration of an informal strategy”, a description of the object is provided. In Activity 4 “formalisation of the solution” the relevant fields of the ODS are filled.
- ***Environment object:*** If the Environment object does not already exist (reuse objects), the provided interface of the ODS should be created in Activity 4.
- ***Class:*** Each Class should be designed separately as a root object of a new system to design outside the design in which its instances are used. There is no basic design step to be performed for an instance of a Class.
- ***Virtual node:*** The basic design step performed for the Virtual Node follows the general schemes of a non-terminal object.

As an illustration of the basic design step activities a simple example is detailed in Section 1.5 below.

1.3.2 THE OVERALL HOOD DESIGN PROCESS

The previous basic HOOD design process allows to build a unique HDT. In order to take into account project management constraints such as subcontracting, parallel development, reuse, system design, etc... The designer has to include new principles allowing to build the whole system configuration, with the different spaces. The overall HOOD design process thus consists in (see also Figure 24 - below):

- defining the system-to-design as an “interface” with respect to its environment
  ☞ **define a root “STD” object and environment objects**

  If one plans to **reuse** objects or classes from a previous project, they will have to be included in the system configuration
  ☞ **define the system configuration**
- performing the first basic design step (decompose the root STD into child objects)
  ☞ **update the system configuration**

  If child objects are similar or reusable,
  If child objects have to be sub-contracted, they will have to be included in the system configuration
  ☞ **update the system configuration**
- iterating basic design steps up to a level of detail enough for direct implementation and coding.

The basic design step is applied in the same way whatever the level of design. It provides activities to be performed by the designer, thus allowing for large systems improvement of the management procedures by allowing the distribution of design and development as well as the definition of milestones providing unique visibility over work progress.

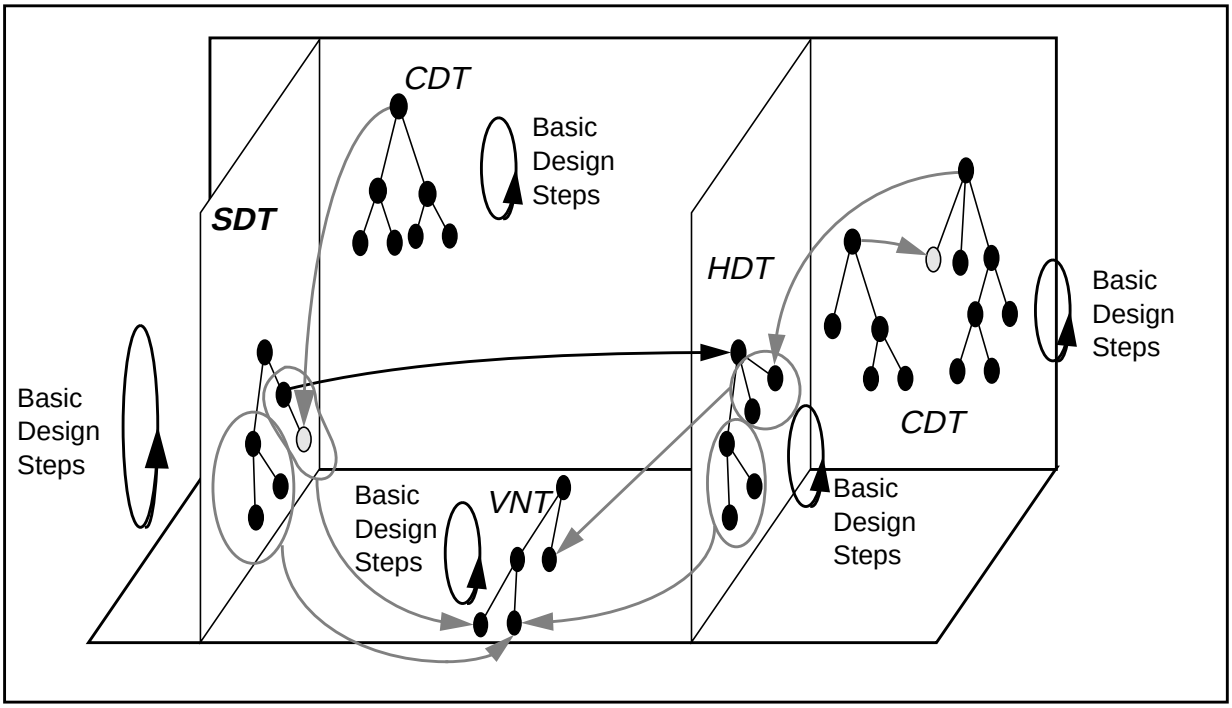


Figure 24 - Application of Basic Design Steps to the system configuration

The general principles of the HOOD design approach are top-down in order to reduce and master complexity. Moreover HOOD offers to a designer a *client-server model* of representation at different levels of abstraction and refinement, through *parent-child composition*, but always keeping consistency with initial representations of the design.
Such properties are exploited in the framework of complex system developments, by producing successive refinements of an initial model down to the operational one with good traceability and high reuse potential. The development approach is thus based on the following principles:

- *Elaboration of an initial model or logical solution.* This model is an abstraction of a solution structured into HOOD object hierarchies, where target implementation related elements are ignored. Such a solution should be fully independent from target and non functional characteristics (languages, targets, efficiency, distribution,...),
- *Refinement of initial models.* HOOD refinement may simply add more details to existing characteristics, or may also add new objects and/or hierarchies of objects in the models, thus allowing refinement trading off with non functional constraints such as existing of-the-shelf software, target constraints, as well as *bottom-up*, *reusing* development approaches.

Although such an approach is the technical way to go, designers are generally reluctant to apply it. They often fear having to break the initial model when they adapt and refine it according to their specific target or project constraints. We believe however that it is always easier to develop a simplified system (and possibly redevelop it) than to start a system integrating all constraints from scratch. Whatever the case, developing an initial HOOD model is efficient in that:

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

The HOOD development approach, producing first logical models reflecting pertinent abstractions of a solution domain, in target independent way, is, at our knowledge, the only one leading to solutions that take into account the following set of constraints:

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

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

- **phase 1: Logical architecture**
  By logical one should understand “non physical” that means that a HOOD design shall first be produced, ignoring all physical and implementation details and constraints. The principle of the approach is to produce first a solution as if non constrained (s*upposing we have an ideal with unlimited power reliable target*) and then rework it to add complexity for dealing with specific non functional constraints such as performance, reliability, distribution....
- **phase 2: Infrastructure**
  By Infrastructure one should understand the support software associated to the logical architecture e.g communication services, operating system, archiving system,...
- **phase 3: Distribution**
  One should understand distributing the software associated to previous architectures and HOOD objects over a physical network of “logical processors”or VNs.
- **phase 4: Physical architecture**
  One should understand here the allocation of the logical processors identified in phase 3 onto the physical architecture consisting of physical processors inter-connected through communication channels.

The four activities of that process are shown in Figure 25 - below:

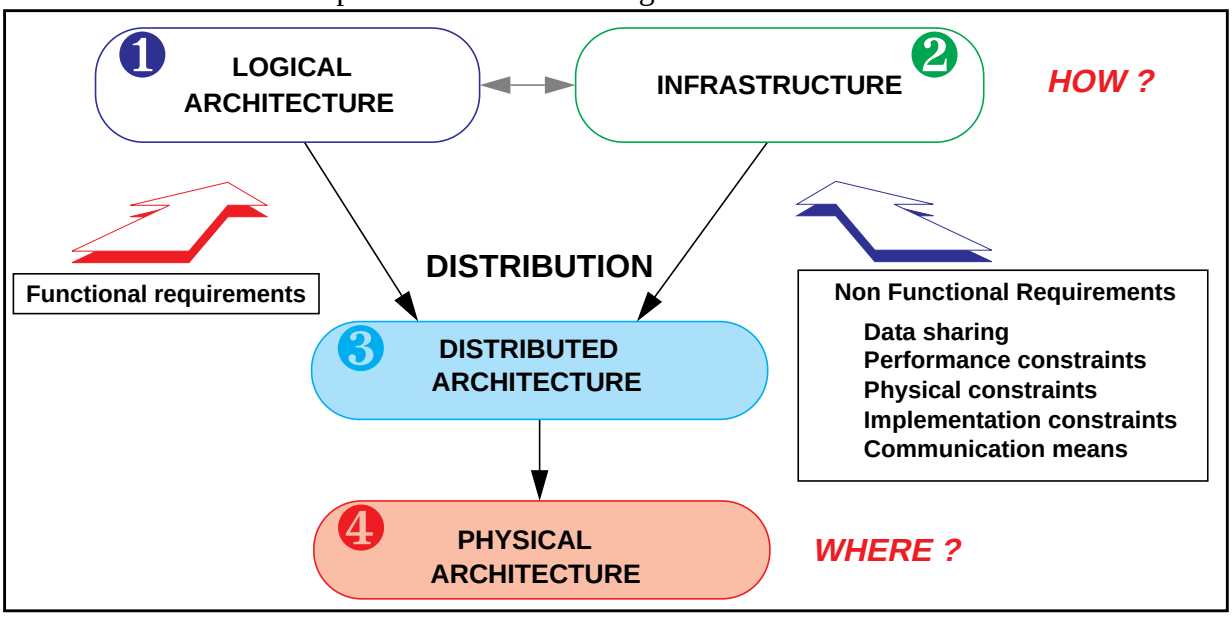


Figure 25 - Full Design Activities Overview

The phase 2 may become insignificant if an existing (OS for example) or reused infrastructure (real time monitor) is used.

These «Full Design Activities» are summarized in Figure 26 - and more detailed in the following sections.

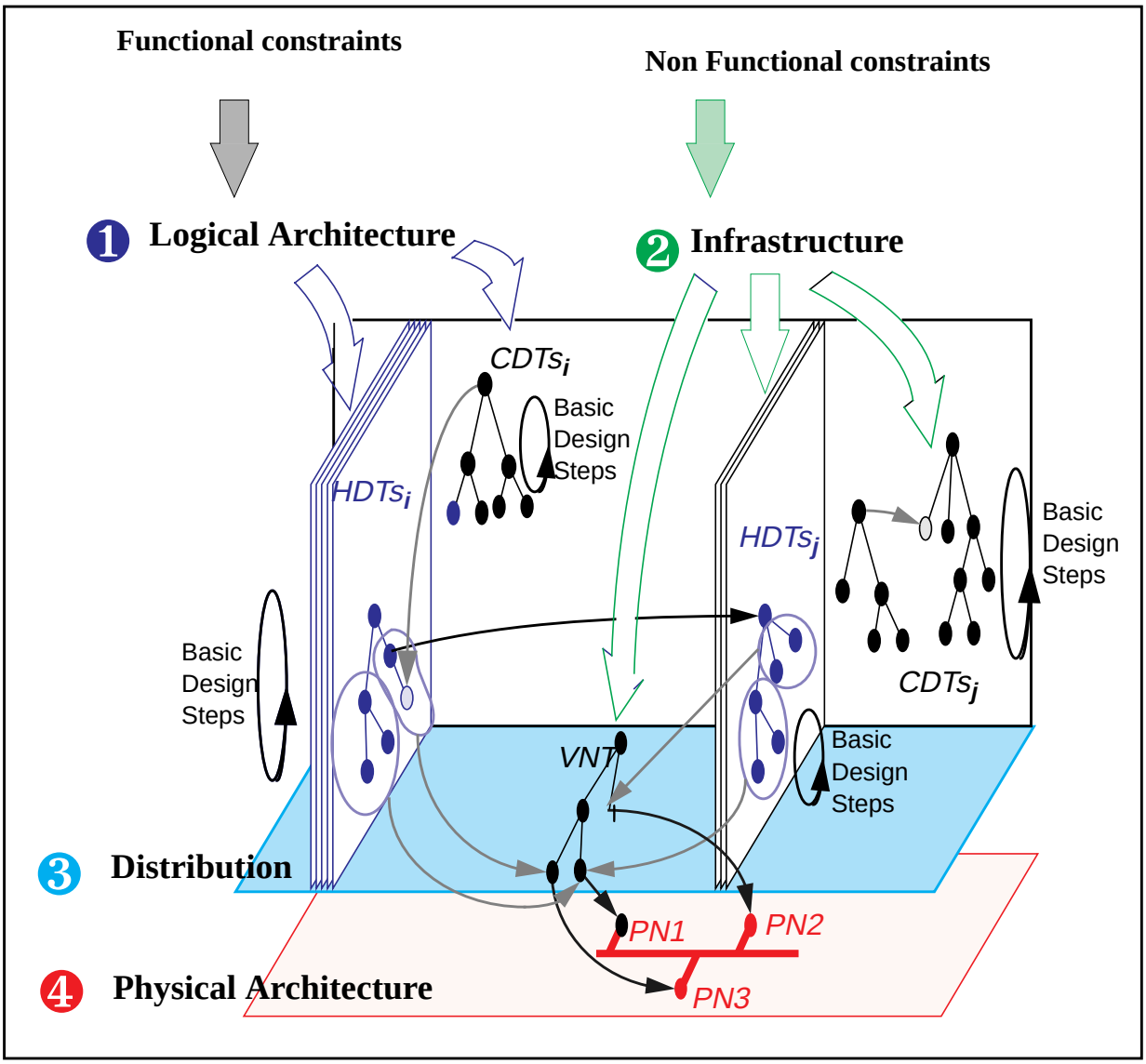


Figure 26 - Full Design Activities applied on the HOOD Architecture

1.3.3 PHASE 1: LOGICAL ARCHITECTURE

This phase is done by the application of HOOD design steps, where the decomposition is produced as a grouping of HOOD objects directly supporting the system to design functionalities, without taking into account the “non-functional constraints”. The resulting design defines the main HOOD Design Tree.
These steps are specification (application) driven. Implementation constraints should not be considered during this phase. At the end of each design steps, similar objects, reusable objects, servers or sub-systems are identified.

Similar objects or reusable objects may be implemented as Classes. Each class is then decomposed by the application of HOOD design steps. The different Class Design Trees are built.

Servers or sub-systems may be implemented as other root objects (environment objects for the main HOOD design tree). These root objects are decomposed into other HDTs.

The corresponding System Configuration is built step by step with the different HDTs and CDTs built during that phase: they constitute the **Applicative Environment** of the system configuration.

1.3.4 PHASE 2: INFRASTRUCTURE ARCHITECTURE

This phase may begin shortly after the logical architecture from the needs identified during that phase or from some non functional requirements such as implementation constraints or performances requirements. It consists of the following activities:

- Creation of the VN tree (not necessary at the beginning). The root VNs may be identified from the analysis of:
  - some project structure constraints such as partitioning into assemblies or sub-systems,
  - some implementation constraints (channel throughout) onto predefined physical nodes.
- Identification of communication protocols needs between the different VNs. These protocols are implemented in specific root objects (HDT) and considered as environment objects for the applicative environment.
- Depending on the project, these root objects are part of the system to design. In that case they are decomposed by the application of HOOD design steps which define new HOOD Design Trees. This decomposition is implementation driven.
- Identification during each design steps of:
  - similar objects or reusable objects; these objects may be implemented into class objects,
  - servers which define new HDTs.

HDTs and CDTs defined during that phase constitute the **Implementation Environment** and the System Configuration is completed accordingly.

1.3.5 PHASE 3: DISTRIBUTION

This phase deals with the distribution aspects. The logical architecture has to be mapped to the infrastructure architecture. This mapping is mainly driven by performance issues such as communication overload, time constraints. It consists in the following activities:

- a further refinement of the VNTs identified in phase 2,
- an allocation of objects from HDTs to terminal VNs of the VNT.

During that phase, the analysis of performance constraints may lead to reallocation of objects onto different VNs or to the creation of new VNs. The analysis of data sharing conflicts may lead to further decompositions of the objects or to allocation of objects in a same VN.
Implementation or logical USE links may also be identified. Interfaces between the different VNs have to be minimized.
During that phase, some modification on the logical architecture can be foreseen mainly to solve data sharing conflicts or to add new objects in the implementation environment.

1.3.6 PHASE 4: PHYSICAL ARCHITECTURE

This phase consists in the allocation of the VNs onto the physical architecture. It consists in the following activities:

- identification of physical nodes (PNs),
- identification of communication channels and of the associated communication protocols,
- allocation of VNs to PNs,
- verification of compatibility between channels and use relation between VNs,
- performance evaluation e.g. estimation of communication overload, response time and throughputs, CPU overload... Specific performance evaluation support such as queuing networks, temporised Petri-nets, worst case execution time estimation techniques...can be used during that phase.

Most of the time, the physical architecture is imposed. In that case, this phase consists only in the allocation of VNs to PNs and in the verification of performances constraints.

1.4 INTEGRATING HOOD IN THE LIFE_CYCLE ACTIVITIES

1.4.1 OVERVIEW

From requirement to design, the HOOD basic design step identifies a sub step which is the analysis and structuring of the requirements. This phase allows to trace the requirements implemented in the design and leads to the concept of “Z” life cycle. In this life cycle, the design starts with a high level specification, then each terminal object is specified before a further breakdown is applied on the object and so on until the detailed design may be performed on the object.

From design to code, the HOOD textual formalism allows possibly automatic translation into Ada C++ or C.

During the design process HOOD can be easily integrated with a reuse environment.

For large projects, HOOD facilitates the share of work and sub contracting by a clear separation of the interfaces of the objects from their internal.

1.4.2 SPECIFICATION TO DESIGN

HOOD is not aimed to support requirements analysis. It has been recognized that methods as IDEF, structured analysis (SA), OMT,OOA... are more suitable to establish the functional breakdown of a system with the customer and user point of view. Therefore, there is the need to establish a mapping between the functional breakdown and the HOOD design objects tree which represents the designer point of view. Unfortunately there is no trivial relationship between both representations.

Although there is no automatic way to find HOOD objects from the functional requirements analysis, some heuristics have been established for the case of the IDEF method:

- the set of IDEF data inputs and outputs may be mapped into the HOOD objects and data flows,
- the set of the IDEF actions may be mapped into the set of operations of the HOOD design.

Other methods such as Petri Nets, State Transition Diagrams are used for Real Time system specification (SA-RT) and may provide essential inputs for describing the behaviour of HOOD active objects (through the OBCS).

For large projects, such as space software projects, the joint use of functional analysis and architectural design approaches is very fruitful.
For such systems it is useful to establish an overall architectural design which needs to be refined with the help of a functional analysis. This is the concept of “Z” strategy which mixes the functional and the architectural analysis (see Figure 27 -). Each terminal object of the level 1, which may correspond to a large subsystem will be decomposed according to a functional analysis with a method as IDEF. Then a further architectural design step is applied on each object. This process is done until the level of complexity of the identified objects allows to start the detailed design. One can see that this strategy is fully compliant with the HOOD basic design step.

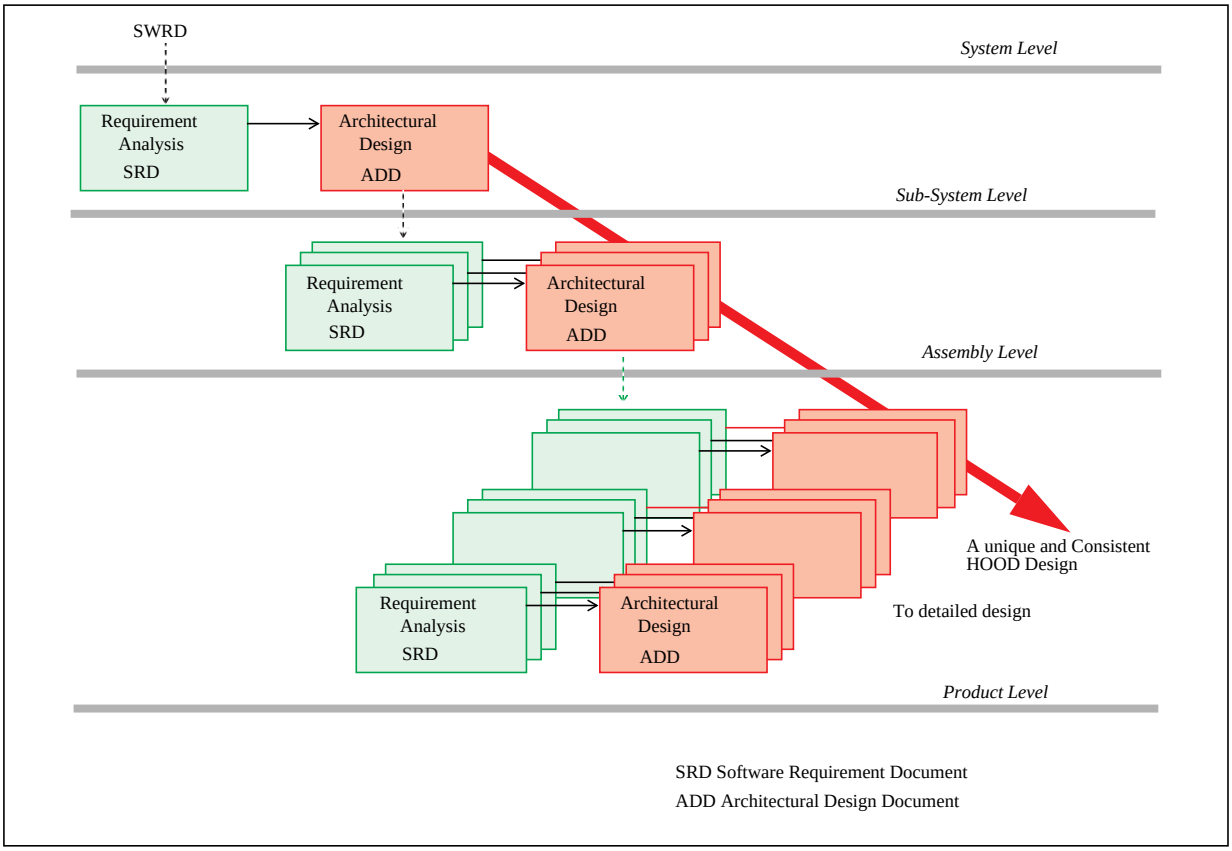


Figure 27 - The "Z" strategy

This strategy leads to establish a set of requirement documents (SRDs) of a manageable size including some independent specification models whereas the HOOD design models included in the different ADDs are completely consistent (Figure 28 -) and parts of the system to design model. The traceability of requirements through the design is much easier as it is made level by level.

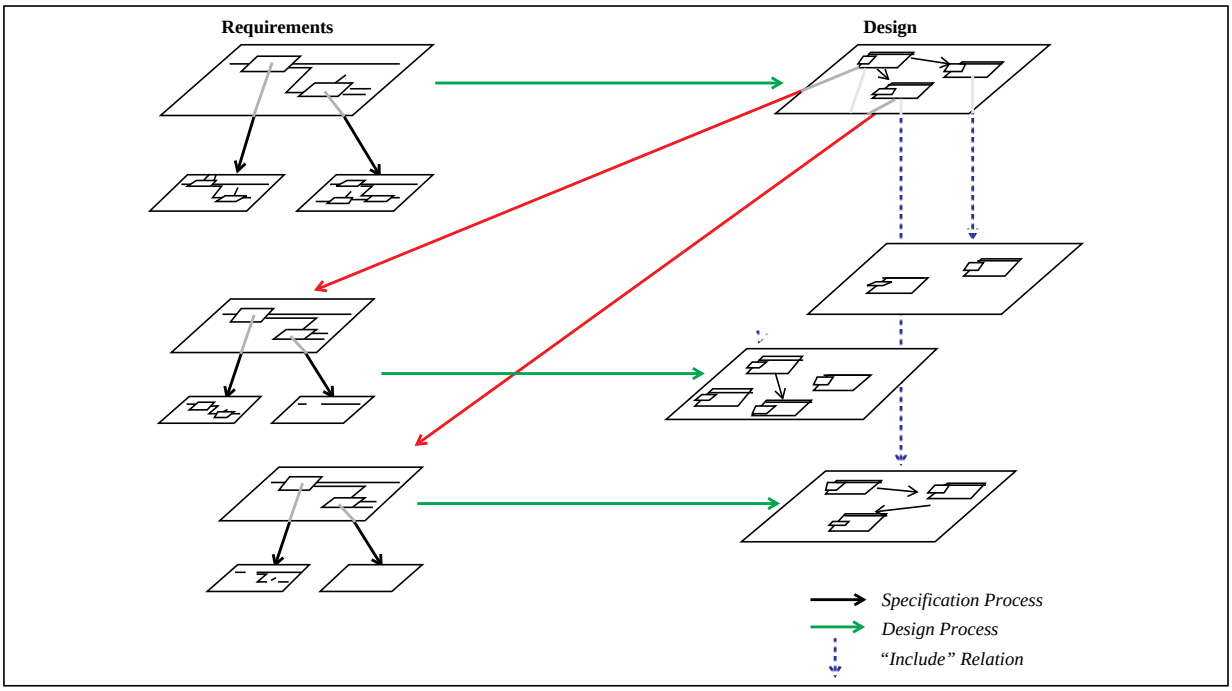


Figure 28 - The different models in the "Z" life cycle

1.4.3 DESIGN TO CODE

HOOD provides translation rules which allow to make more easier the design to code mapping, specially when the coding language is Ada. HOOD does not remove the programming phase but helps the programmer by giving him the possibility to concentrate on local algorithmic and coding problems within the scope of well-defined Ada units whose the interfaces and the body skeleton are directly derived from the design objects.
The apparent equivalence of the concepts in the HOOD method and Ada constructs does not imply passive object and active object are just another word for package and task. Indeed, objects are implemented by an Ada package, but in a very well-defined form with precise properties.
When the coding is not done in Ada, specific translation rules must be defined for the target language but the overall approach remains the same.
A set of PRAGMAs is defined to give the designer a better control of the translation scheme. This allows a development process in which the detailed design is a stepwise refinement of the architectural design then the code editing is the refinement of the detailed design. So the designer and the coder see only the HOOD structure and the ODSs, the final translation into Ada being done just at the end of the code phase or for design prototyping purposes.
With a such approach, code and design are always consistent

1.4.4 TESTS AND VALIDATION

1.4.4.1 Unit and integration tests

The unit and integration tests of parts of code developed using HOOD are done respectively against the detailed design and the architectural design. The HOOD object concept has obvious benefits for testing:

- from the definition of object interfaces (provided/required) it is possible to set-up a test environment
- information hiding inherent to objects, insures to have limited interaction with the test environment

A typical HOOD object test environment (Figure 29 -) will be composed of:

- a test monitor in charge to trigger accessible operations defined in the provided interface
- a test simulator in charge to simulate the operations defined in the required interface

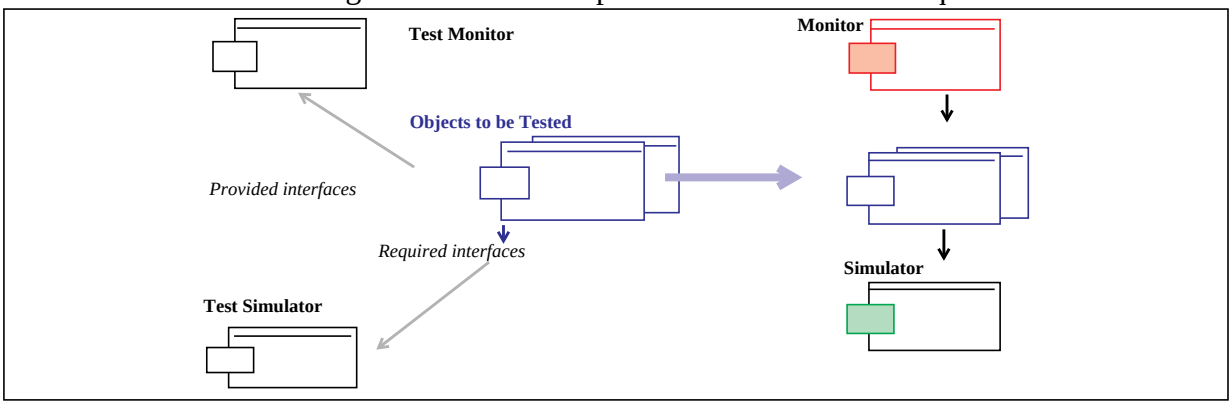


Figure 29 - Test environment generation

The relevant code for the test monitor and test simulator may be automatically generated from the analysis of the design.
Thus, HOOD may support several testing strategies:

- module testing is defined as the activity of validating the behaviour of an operation with respect to its interface specification.
- a bottom-up test strategy would start with objects that are only used
- a top-down test strategy would require main control flows to be completed, and stubs provided for other operations
- a sub-tree approach would allow parts of the HOOD hierarchy to be developed as sub-trees, providing successively more functionality across the design.

1.4.4.2 Verification and Validation

One of the major problem encountered during the development of RT&E (Real Time and Embedded) systems is the V&V (Verification and Validation) activity during all the life cycle phases in order to detect errors very early during the development. Early verification allows to go further in the development steps from a consolidated and validated basis. A key feature of V&V activity is the verification of the dynamic behaviour of the system under design.
Techniques in software verification can be classified into two categories:

- **Verification by simulation** is running a system or a subset of the system with a selected set of input data and evaluates its response. Testing and/or prototyping are verification by simulation.
- **Verification by proofs** is making some mathematical proofs on intrinsic properties of a system and/or on the behaviour invariants of a system when going toward its final implementation. These proofs can be done only if one uses a formal language, which allows logic calculi.

These two kinds of verification are complementary ones. Simulation is interesting to reveal unexpected behaviour of a system, clean up bugs and get a better understanding of the system. Completely proven developments are quite too costly to put into practice, but must be planned for critical RT&E systems.

The HOOD design process decomposes the design activity into basic design steps which facilitates the implementation of the V&V activities. After each step, a validation of the design is possible through informal techniques such as inspection, walk-through and reviews or verification techniques such as design prototyping and design simulation.
Design validation and verification procedures in HOOD should be based on the separation (in time) between the external description of an object and the availability of its implementation.
Two complementary activities are performed:

- step verification, ensuring consistency of one object decomposition
- level verification and validation, ensuring consistency of a complete level decomposition (i.e. several objects together).

Step verification mainly consist of:

- checking the object interfaces,
- checking parent-child signatures,
- tracing the mapping of requirements behavioural models into Object Control Structure descriptions.

Level validation is performed in order to check the consistency over several design steps together. The definition of a design prototyping by implementing each terminal object allows to set-up a test and pre-integration environment for each object with respect to its brothers.

1.4.5 HOOD AND REUSE

Reusability is today a topic of primary practical importance. In this area it appears that, reusability of design is the most realistic approach since reuse of code is often to much target computers and data management systems dependant
The use of HOOD greatly facilitates the software reusing process at the design stage. Studies are being conducted in the ESPRIT and EUREKA research programs to enhance the strategy of design in that direction and to develop relevant environments.
Reuse consists of two major activities:

- design a current system to be reusable in other projects
- reuse parts of previous projects during the current system design

HOOD is a globally top-down process while reuse is globally bottom-up. So the reuse process is included in the basic design step by the definition of different scenarios for reuse of design. After the formalization of the solution, one can identify the possible areas in which reusable designs i.e. objects could be reused. Then, the designer researches in a component library the objects which could solve his problem. This search gives a short list of candidates and a “design to target” tries to include those candidates in the current design. This process is similar to the design process for hardware components including commercial chips (IC, VLSI...).
The encapsulation principles in HOOD fit quite well on this scenario in the sense that the user does not need to know the internals of an object and is only concerned with its provided interface which describes how to use the object and its required interface which describes the context and the environment requested by the object to provide the services.
More details on these topics are given in Section 2.10 of this document.

1.4.6 HOOD AND SUBCONTRACTING

There is a clear need to properly manage SW development in the context of large systems involving many contractors and subcontractors. In this context, HOOD may be used according either to the “prime” point of view or to the “subcontractor” point of view.
The “prime” HOOD activities are mainly concerned with:

- responsibility of high-level design,
- interface validation,
- identification of HOOD sub-trees, subject to subcontracting,
- prototyping activities, by simulating subcontracted software parts mainly to validate the dynamical behaviour of the system,
- integration by replacing simulated code by delivered final code.

The “Subcontractor” HOOD Activities Are Mainly Concerned With:

- responsibility of a complete hood sub-tree,
- internal design activity performed according to the interfaces defined, validated and agreed with “prime”,

- possible pre-integration activity.

1.4.7 PHASED INCREMENTAL LIFE CYCLE

On large projects, requirements definition is a *non monotone function of the development process:* requirements become more and more precise and mastered as the project goes forward. Large projects developments have been compared to “cathedral building” and such top-down development are now more and more questioned.

The risk of such projects can however be limited and at the same time conforming their development to an organisational (management model) model of the V cycle, by splitting the project into phases[12], each of which is conducted as a subproject with deliverables and possibly an own V life-cycle. In order to achieved some efficiency inputs of a phase may be taken from outputs of a previous one.

The way the initial partitioning is done is of course of prime importance for the success of the project, but hood recommends to follow a phased approach in order to limit this risk:

- start first by defining a logical partitioning that highlights relevant abstractions of the solution (phase 1),
- refine down the logical model and start to build the infrastructure model and the distribution model to a verifiable, executable model on a ideal target (phase 2),
- adapt, refine further and tune on the final target by refinement of both model taking into account the physical model (phase 3).

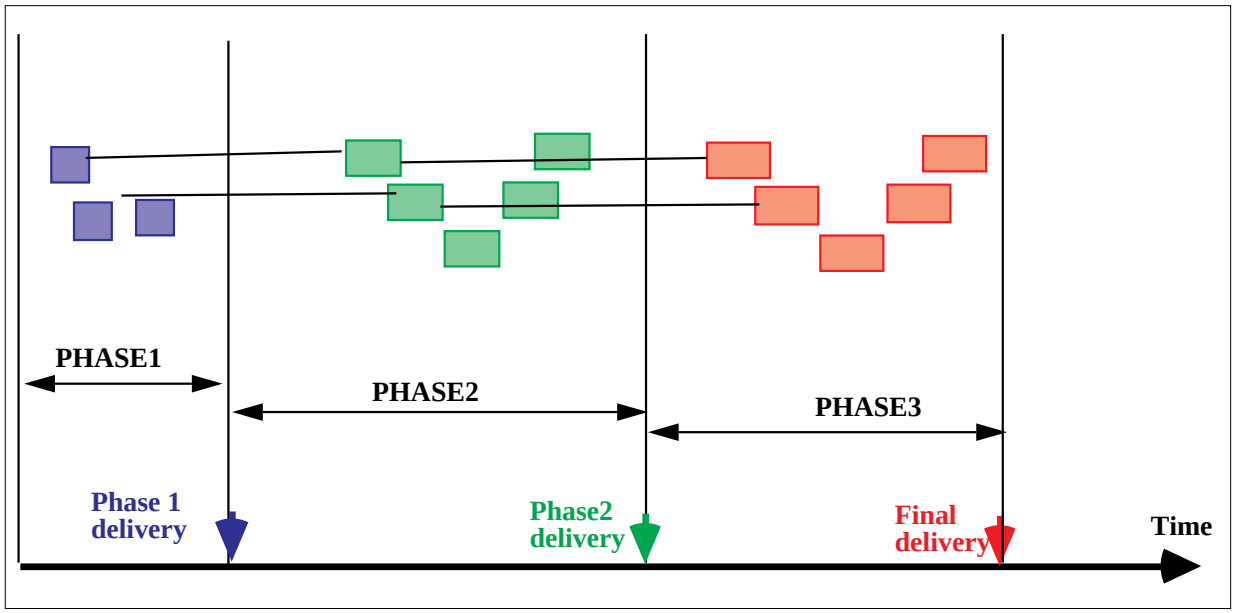


Figure 30 - Phased Incremental Development Approach For Complex Systems

[12] Experience has shown that an “appropriate” duration for such phases should be less than one year.

1.5 A HOOD EXAMPLE

In this section we shall illustrate a HOOD design, not a full fledged one, but trying merely to illustrate a full design step.

1.5.1 PRESENTATION OF THE EMS SYSTEM

The EMS is a software to monitor car engine parameters , to display them on a bargraph and trigger an alarm if associated values are out of range. A detailed requirement of the EMS is given in *APPENDIX A1* - of this document.

1.5.2 EMS SOLUTION

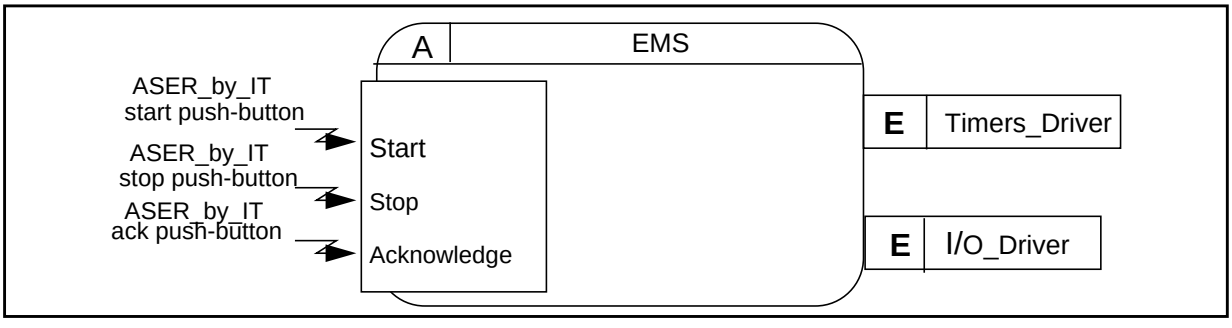
A solution of the EMS software is defined by applying a basic design step.

1.5.2.1 Statement of the Problem (H1.1)

Design and develop a software to controls sensors and display associated values of a car engine.

1.5.2.2 Analysis and Structuring of Requirement Data (H1.2)[1]

- **Analysis and definition of the EMS Environment**:
  The EMS software is running on the CPU, using an Input/Output and a Timers drivers.
  The EMS is defined as an interface to its environment:
  - Start, Stop Acknowledge are provided operations which are triggered by IT (handled by the ADA Real Time Monitor) as the driver/operator will push the associated buttons;
  - The interface to the ADA RTS is represented as two environment objects associated to the TIMER and the I/O boards



1. Questions raised and answered during this activities

How to take into account the hardware interfaces (from push-buttons, sensor failure...)?
Which is the EMS software environment? Which is the range for oil, fuel and water? How to display a fault (sensor failure or value out of range)? How to take into account several sensor failures or several values out of range?
How to stop the Alarm (with Ack push button, when fault has disappeared)? What about an existing Alarm after a start? How and When to start and stop the system? The Acknowledge shall stop the Alarm for all faults which have set it.

- **Analysis of Functional Requirements:**
  The function to performed by the EMS are:
  -acquire the data from the sensors ==> **handle ACQUISITION**
  -compute the mean value
  -compare it the limit values of pressure, temperature and level ==> **handle LIMITS**
  -display the values in green if ok, in red if not, flashing it failure ==> **handle DISPLAY**
  -trigger the alarm ==> **handle the ALARM**
  - handle sensors or display faults ==> **handle the ERRORS**
- **Analysis of Behavioural Requirements:**
  The EMS can be started and stopped, at any time, independently of the car engine.The EMS can be started and stopped at any moment by pushing the associated push buttons, independently of the car engine.
  - A start when the EMS is running shall have no impact on the EMS.
    A stop when the EMS is stopped yet shall have no impact on the EMS.
    An acknowledge when the alarm is switched off or the EMS stopped shall have no impact on the EMS.
  - When the Alarm is set, it remains displayed until the ack push button is pushed or until the corresponding fault has disappeared.
    The Alarm shall be set only for a new fault (when the same fault has previously disappeared).
    Alarm is restarted after a start (an ack before the previous stop shall be ignored).

Figure 31 - below summarizes the behaviour of the EMS system.

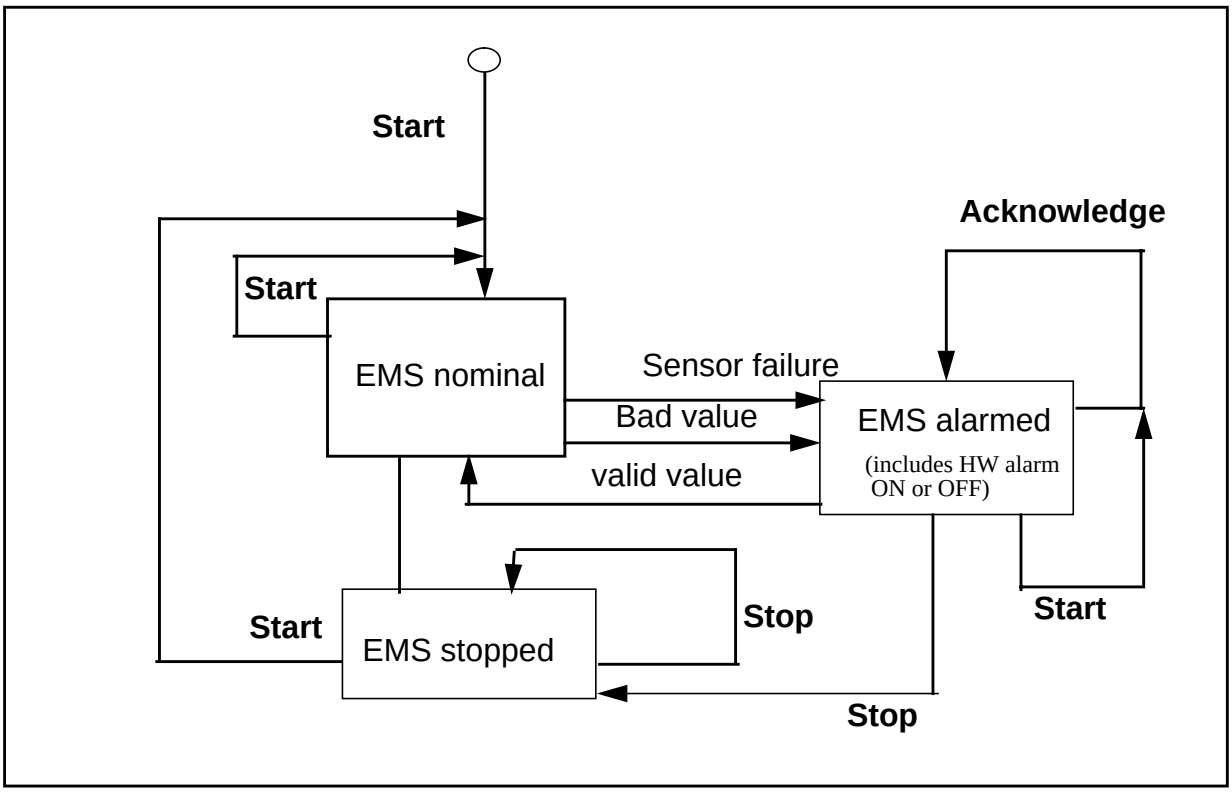


Figure 31 - State Transition Diagram modelling the behaviour of the EMS system

- **Analysis of Non-Functional Requirements**:
  - The EMS must be reliable. More sensors might be handled. The EMS system might be connected to a general vehicle control system.
  - The values shall be smoothly displayed on the bargraphs (particularly in case of sensor failure). Thus the display device shall continuously evolve.
- **User Manual Outline**
  The values of water temperature, oil pressure and fuel level are displayed in green light on three different bargraphs. In case of sensor failure, the corresponding bargraph flashes in red light and the alarm is started. In case of a value is out of range, the bargraph displays the value in red light and triggers the alarm.
  The EMS can be started and stopped by using start and stop push buttons.
  The alarm of the EMS may be switched off by the ack push button.

1.5.2.3 Informal Solution Strategy[2] (H2)

On IT reception from the Start push button, the EMS is started: the Ctrl_EMS inits the bargraphs, starts the alarm and the sensors and creates and starts a timer which triggers the monitoring each second (monitoring timer). Initialisation of bargraphs switches on the hardware bargraph through the I/O driver. Starting the alarm switches off the hardware alarm through the I/O driver. Starting the sensors starts the hardware sensors and creates and starts a timer which triggers the sampling at 10 Hz (sampling timer).

When the sensors are sampled every 1/10 second by a signal from the timer, the values and the status (malfunction) of each sensor are get from the hardware sensors (through the I/O driver).
Every second, the monitoring activity is triggered by a signal from the timer: the Ctrl_EMS acquires the status and the mean values of oil pressure, fuel level and water temperature from sensors. It compares them with the limited values.

If a mean value is out of range or a sensor has failed, the Ctrl_EMS switches the alarm on (with the type of sensor and the type of failure). If there is no failure and the mean value is correct, it displays each mean value on the appropriate bargraph, sets the colour in green, stops flashing and switches the alarm off. If a mean value is out of range, the corresponding bargraph is displayed in red light. If there is a sensor failure, the corresponding bargraph flashes in red light. An eventual malfunction of the hardware bargraphs or the hardware alarm is of no effect on the Ctrl_EMS.

On IT reception from the Ack push button, the alarm is acknowledged.
On IT reception from the Stop push button, the EMS is stopped if it was not: the alarm, the monitoring timer and the sensors are stopped and the bargraphs are switched off. Switching the bargraphs off stops the hardware bargraphs (switches light off and put values to zero through the I/O driver). Stopping the alarm stops the hardware alarm through the I/O driver. Stopping the sensors stops the hardware sensors (through the I/O driver) and stops the sampling timer.

[2]. Note that this strategy is the ultimate one found after refinement of earlier outlines as the candidate child objects and associated operations were described and refined in «Formalization of the Strategy activities»

1.5.2.4 Formalization of the Strategy (H3)

1.5.2.4.a Identification of objects (H3.1)

Object BARGRAPHS:
The bargraphs object allow to display values, in red or green with or without flashing, on appropriate display devices.It also provides all means to start and stop the hardware display device.

Object CTRL_EMS:
This object is the controller of the EMS. It starts and stops all the constituents objects of the EMS and controls the monitoring.
The Ctrl_EMS inits the bargraphs, starts the alarm and the sensors and creates and starts a timer which triggers the monitoring each second (monitoring timer).

Object SENSORS:
This object samples oil pressure, water temperature and fuel level at 10Hz and stores the read values of the three sensors at any moment. It may provide the mean of stored values of a sensor. It also provides all means to start and stop the hardware sensors.

Object ALARM
The Alarm object manages a set of software alarms. One hardware alarm is associated to those software alarms. It is possible to switch a software alarm on or off at each time. The hardware alarm is started when an unset software alarm is switched on (set). The hardware alarm is stopped when the set of software alarms are switched off (unset) or when acknowledged by the user. In that case the status of the set software alarms are not modified.

1.5.2.4.b Identification of operation (H3.2)

Object Input_Output_Driver:

- **Put** The Put operation allows to write information included in the provided descriptor to the hardware device corresponding to the specified one.
- **Get** The Get operation allows to copy the current information, provided by the hardware device corresponding to the provided one, into the provided descriptor.

Object CTRL_EMS:

- **Start** creates the monitoring timer, initialises the alarm, the sensors and the bargraphs which are used during the monitoring.
- **Stop**; stops the monitoring timer and switches off the alarm and the bargraphs and stops the sensors.
- **Monitor**; acquires the values of each sensor. Those values are displayed on the appropriate bargraph. In case of a value is out of range, this operation switches the alarm on and gives the red colour to the bargraph. If there is a sensor failure, the alarm is switched on and the bargraph flashes in red colour. In other cases, the values are displayed in green colour.

Object SENSORS:

- **Start**; initializes the hardware sensors.
- **Sample** is in charge of the Sensors sampling. It gets the current values of each hardware sensor and stores them into an internal database.
- **Acquire** returns the mean of the ten last stored values of a sensor. It returns a

SENSOR_FAILURE exception in case of hardware sensor problem.

- **Stop** stops the hardware sensors.

Object BARGRAPHS:

- **Init**; initializes the hardware bargraphs. Bargraphs are switched on in green colour without flashing and with a null value.
- **Display** displays the input percentage value in the corresponding hardware bargraph. The colour and the flashing are not modified.
- **Set_Colour** sets the corresponding hardware bargraph in the specified colour. The displayed value and the flashing are not modified.
- **Flash** allows to flash the corresponding hardware bargraph. The colour and the displayed value are not modified.
- **Switch_Off** switches the hardware bargraphs off.

Object ALARM:

- **Start** switches the hardware alarm and the software alarms off.
- **Acknowledge** stops the hardware alarm without unset the set software alarms.
- **Switch_On** switched the specified software alarm on. If this alarm was not yet set, the hardware alarm is started (if it was not). In the other cases, nothing is modified.
- **Switch_Off** The Switch_Off operation switched the specified software alarm off. If the hardware alarm is yet started and all the other software alarms are unset, the hardware alarm is stopped.
- **Stop**; switches the hardware alarm off.

1.5.2.4.c HOOD DIAGRAM of EMS DECOMPOSITION (H3.4)

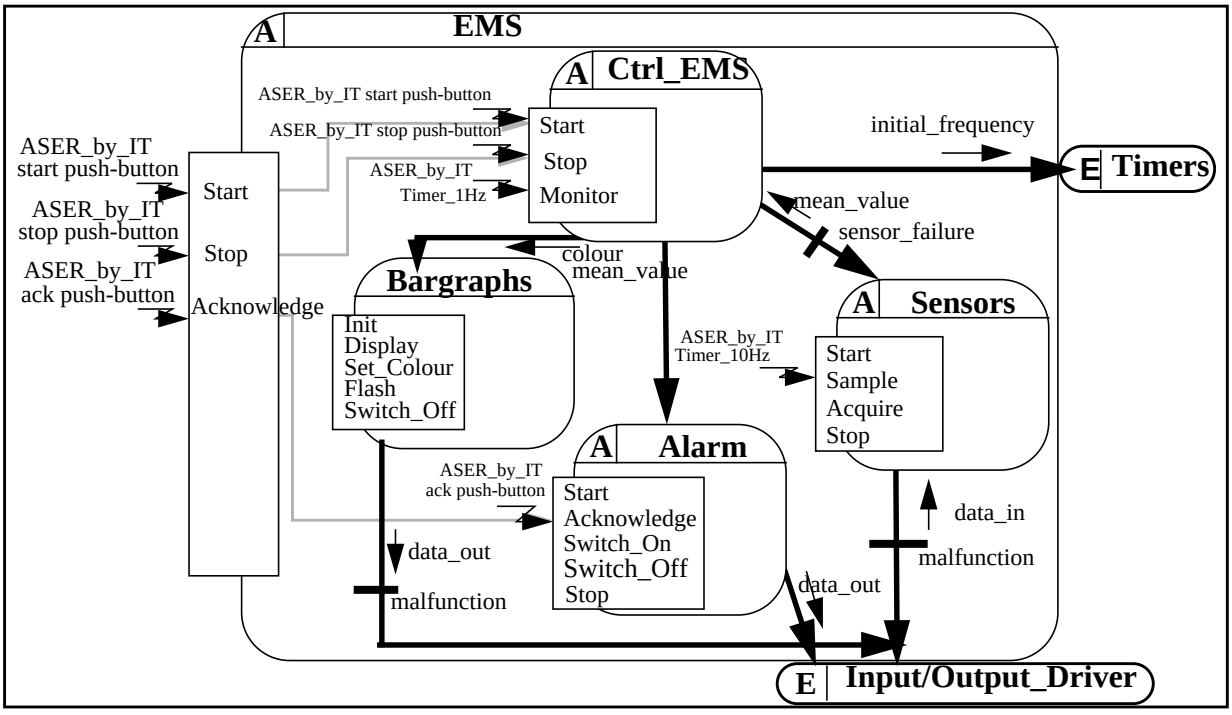


Figure 32 - EMS HOOD DIAGRAM (H3.4)

1.5.2.4.d Justification of Design Decisions (H3.5)

Object BARGRAPHS implements the display functional constraints. It is able to display additional values.
Object ALARM implements the interface with the alarm system. Any change in the alarm system will only affect the alarm object.
Object SENSORS handles the acquisition function, it is designed so as to be able to handle additional sensors.
Object CTRL_EMS handles initialisation and stop of timers interrupts, as well as push button It.

1.5.2.4.e Grouping Operations & Objects (H3.3)

OBJECTS	OPERATIONS	COMMENTS
EMS		
	Start	On IT from Start push-button
	Stop	On IT from Stop push-button
	Acknowledge	On IT from Ack push-button | implemented by ALARM
Bargraphs		
	Display	value (oil, water, fuel) : IN
	Set_colour	colour (red, green) : IN
	Flash	
	Switch_off	
Sensors		
	Sample	value (oil, water, fuel) : OUT | every 1/10 second Tim
	Acquire	value (oil, water, fuel) : OUT
	Stop	
		sensor_failure : exception
Alarm		
	Switch_on	
	Stop	
	Acknowledge	On IT from Ack push-button | implements EMS Reset
Ctrl_EMS		
	Monitor	every second by Timer signal
	Is_value_out_of_range	internal operation|
		value : IN limited_oil_value : CONST, limited_water_value : CONST, limited_fuel_value : CONST

This section is given for illustration only. Its added value is a matter of discussion and taste and depends highly of the toolset being used, experience, and project reviewing standards.

1.5.2.5 Structuring the design

Analysis of reusable objects,
Analysis of reused object,
Analysis of distribution aspects,
.....
=> Building system configuration

```
SYSTEM_CONFIGURATION
        ROOT OBJECTS
                EMS, Timers_Driver, IO_Driver
        CLASSES
                Sensors, Bargraphs
END
```

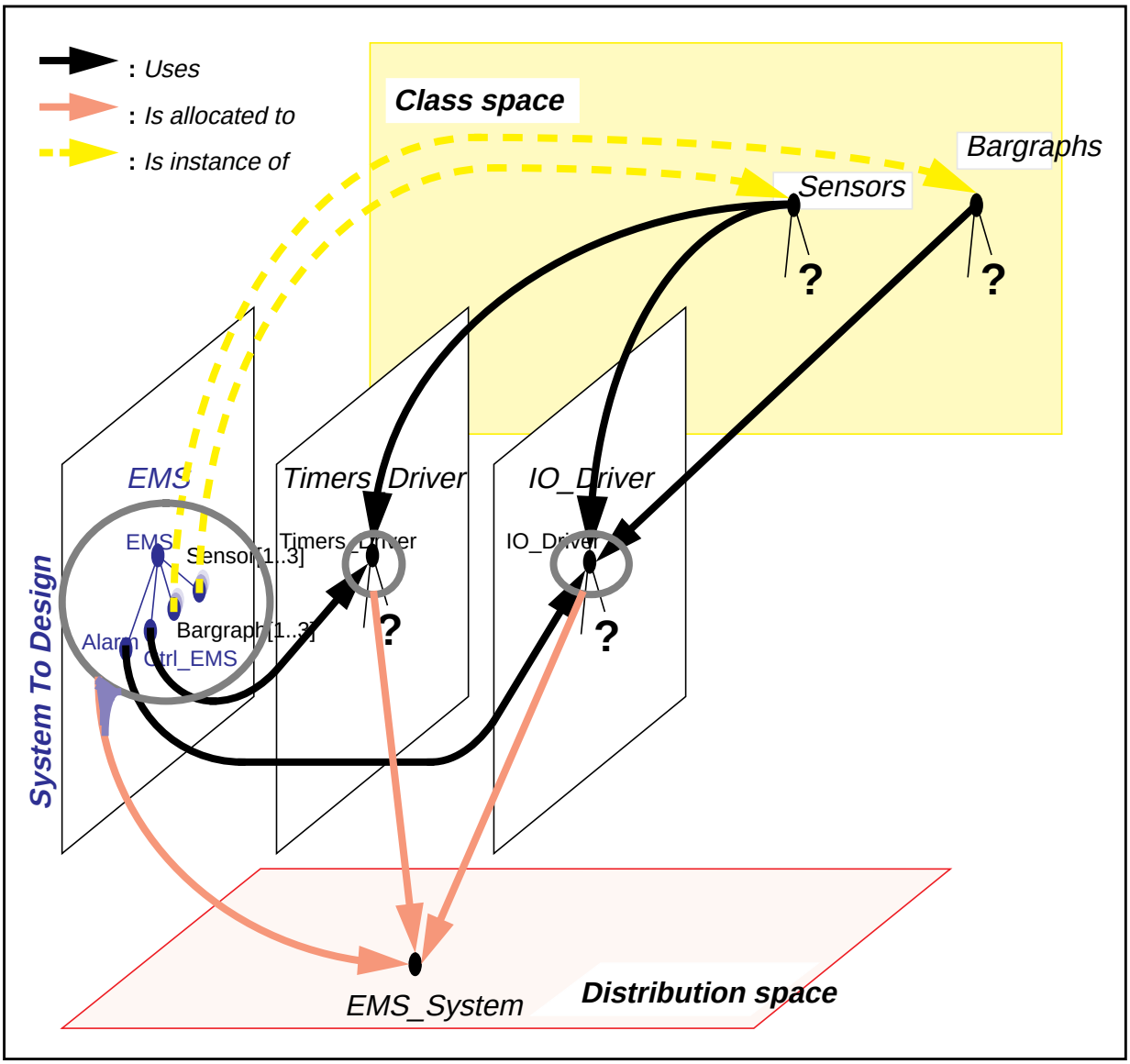


Figure 33 - EMS objects and Design Views

1.5.3 ODS EXAMPLES OF EMS SYSTEM

1.5.3.1 ENVIRONMENT OBJECT Input_Output_Driver

OBJECT Input_Output_Driver **is ENVIRONEMENT PASSIVE**
DESCRIPTION
The Input_Output_Driver object provides means to exchange information with hardware devices. It allows to put or to set information of these devices.
IMPLEMENTATION_CONSTRAINTS NONE
PROVIDED_INTERFACE
CONSTANTS
MAX_BARGRAPHS_NUMBER: **constant** := 3;
Number of hardware bargraphs managed by the Input_Output_Driver object.
MAX_SENSORS_NUMBER : **constant** := 3;
Number of hardware sensors managed by the Input_Output_Driver object.
MAX_ALARMS_NUMBER : **constant** := 1;
Number of hardware alarms managed by the Input_Output_Driver object.
TYPES
T_DEVICE **is**(ALARM, BARGRAPH, SENSOR);
The T_DEVICE type defines the set of devices which are taken into account by the Input_Output_Driver.
T_DEVICE_STATUS **is** (ON, OFF);
Defines the status of a a device. ON : the hardware device is running, OFF : the hardware device is stopped.
T_BARGRAPH_COLOUR **is** (RED, GREEN);
This type defines the two different colours of a bargraph.
T_BARGRAPH_FLASH_STATUS **is** (ON, OFF);
This type defines the two different displaying status of a hardware bargraph.
T_DESCRIPTOR (DEVICE : T_DEVICE) **is**
record
STATUS : T_DEVICE_STATUS := OFF;
case DEVICE **is**
when BARGRAPH =>
NUMBER : INTEGER **range** 1 .. MAX_BARGRAPHS_NUMBER := 1;
VALUE : INTEGER **range** 0 .. 100 := 0;
COLOUR : T_BARGRAPH_COLOUR := GREEN;
FLASH : T_BARGRAPH_FLASH_STATUS := OFF;
when SENSOR **=>**
NUMBER : INTEGER **range** 1 .. MAX_SENSORS_NUMBER := 1;
VALUE : INTEGER **range** 0 .. 100 := 0;
when ALARM =>
NUMBER : INTEGER **range** 1 .. MAX_ALARMS_NUMBER := 1;
end case;
end record;
The T_DESCRIPTOR type defines the set of information which may be sent to output devices or received from input devices.
OPERATIONS
Put(DEVICE : **in** T_DEVICE; -- Output Device
ELEMENT : **in** T_DESCRIPTOR -- Descriptor to apply to a device);
The Put operation allows to write information included in the provided descriptor to the hardware device corresponding to the specified one.
Get(DEVICE : **in** T_DEVICE; -- Input Device
ELEMENT : **out** T_DESCRIPTOR -- Descriptor including information from device);
The Get operation allows to copy the current information, provided by the hardware device corresponding to the provided one, into the provided descriptor.
EXCEPTIONS
MALFUNCTION **RAISED_BY** Put, Get;
This exception is returned when the hardware device specified in the Put or Get operations is unavailable.
END_OBJECT *Input_Output_Driver*

1.5.3.2 PARENT OBJECT EMS

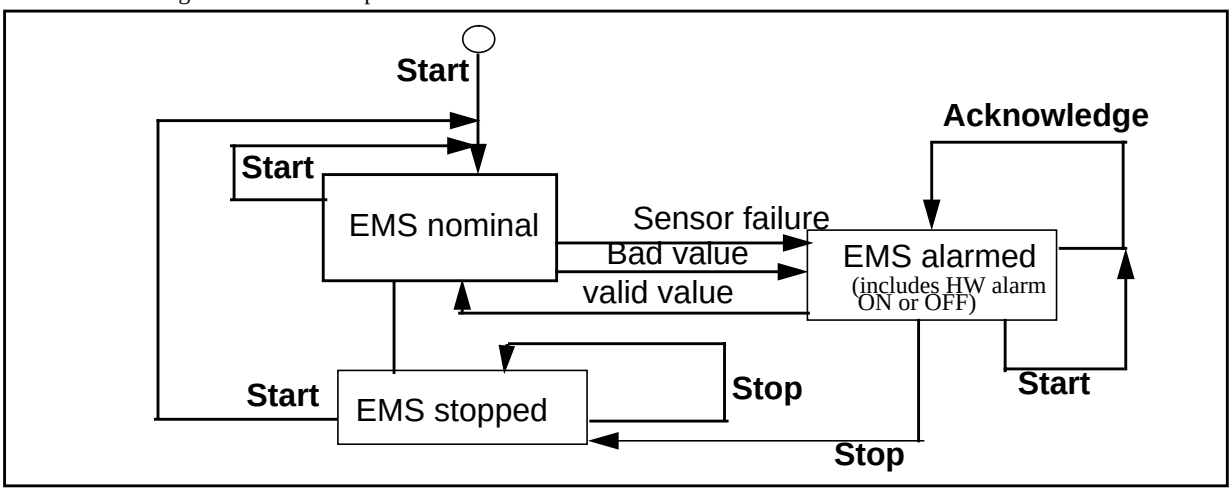
OBJECT EMS **IS ACTIVE**
DESCRIPTION
The EMS monitors and displays the oil pressure, water temperature and fuel level on the appropriate bargraphs, and starts an alarm in case of a value is out of a predefined range. First, the EMS shall be started.
IMPLEMENTATION_CONSTRAINTS
Start, Stop and Acknowledge operations are mutually exclusive (each of those operations shall wait the end of an other one before to be started).
PROVIDED_INTERFACE
OPERATIONS
Start;
The Start operation starts the EMS. The EMS environment will be initialised.
Stop;
The Stop operation stops the EMS.
Acknowledge;
This operation stops the Alarm for the faults which have switched on it.
EXCEPTIONS
NONE
OBJECT_CONTROL_STRUCTURE
DESCRIPTION
The EMS accepts start, stop and acknowledge operations at any time. Start is not significant after a start. Stop or acknowledge are not significant after a stop.

CONSTRAINED_OPERATIONS
Start CONSTRAINED_BY ASER_by_IT Start_push-button;
Stop CONSTRAINED_BY ASER_by_IT Stop_push-button;
Acknowledge CONSTRAINED_BY ASER_by_IT Ack_push-button;
REQUIRED_INTERFACE

INTERNALS
OBJECTS
Ctrl_EMS;
Sensors;
Bargraphs;
Alarm;
OPERATIONS
Start **IMPLEMENTED_BY** Ctrl_EMS.Start;
Stop **IMPLEMENTED_BY** Ctrl_EMS.Stop;
Acknowledge **IMPLEMENTED_BY** Alarm.Acknowledge;
EXCEPTIONS
NONE
OBJECT_CONTROL_STRUCTURE
IMPLEMENTED_BY Ctrl_EMS, Alarm;
END_OBJECT *EMS*

1.5.3.3 TERMINAL CHILD OBJECT CTRL_EMS

OBJECT Ctrl_EMS **IS ACTIVE**
DESCRIPTION
This object is the controller of the EMS. It starts and stops all the constituents objects of the EMS and controls the monitoring
-IMPLEMENTATION_CONSTRAINTS
The Ctrl_EMS must perform the monitor operation in less than 1 second.
PROVIDED_INTERFACE
TYPES NONE
CONSTANTS NONE
OPERATION_SETS NONE
OPERATIONS
Start;
The Start operation creates the monitoring timer, initialises the alarm, the sensors and the bargraphs which are used during the monitoring.
Stop;
The Stop operation stops the monitoring timer and switches off the alarm and the bargraphs and stops the sensors.
Monitor;
The Monitor operation acquires the values of each sensor. Those values are displayed on the appropriate bargraph. In case of a value is out of range, this operation switches the alarm on and gives the red colour to the bargraph. If there is a sensor failure, the alarm is switched on and the bargraph flashes in red colour. In other cases, the values are displayed in green colour.
EXCEPTIONS
NONE
OBJECT_CONTROL_STRUCTURE
DESCRIPTION
The Ctrl_EMS accepts start, stop and monitor operations at any time. A start is not significant after a start. A stop or a monitor are not significant after a stop.



Start CONSTRAINED_BY ASER_by_IT Start push-button;
Stop CONSTRAINED_BY ASER_by_IT Stop push-button;
Monitor CONSTRAINED_BY ASER_by_IT Timer_1Hz;
REQUIRED_INTERFACE
OBJECT Sensors;
TYPES
T_SENSOR;
OPERATIONS
Start;
Acquire;
Stop;
EXCEPTIONS
SENSOR_FAILURE;
OBJECT Alarm;
TYPES
T_ALARM;
OPERATIONS

```
        Start;
        Switch_On;
        Switch_Off;
        Stop;
OBJECT Bargraphs;
    TYPES
        T_COLOUR;
        T_BARGRAPH;
        T_FLASHING;
        T_PERCENTAGE;
    OPERATIONS
        Init;
        Display;
        Set_Colour;
        Flash;
        Switch_Off;
OBJECT Timers_Driver;
    TYPES
        T_TIMER;
    OPERATIONS
        Create;
        Start;
        Delete;

DATAFLOWS
    mean_value <= Sensors;
    mean_value => Bargraphs;
    colour => Bargraphs;
    initial_frequency => Timers_Driver;

EXCEPTION_FLOWS
    sensor_failure <= Sensors;
```

INTERNALS

OBJECTS
NONE
TYPES
TBD
CONSTANTS
IT_1HZ_ADDRESS : constant := Monitor'ADDRESS;
MONITORING_FREQUENCY : constant :=1;
TBD
OPERATIONS
Is_Value_out_of_Range;
TBD
EXCEPTIONS
TBD
DATA
MONITORING_TIMER : Timers_Driver.T_TIMER;
TBD

OBJECT_CONTROL_STRUCTURE
PSEUDO_CODE
TBD
CODE
TBD

OPERATION_CONTROL_STRUCTURES

OPERATION Start
DESCRIPTION
This operation initialises the system (the alarm, the bargraphs, the sensors) and then creates and starts a timer for it triggers the monitoring every 1 second.
USED_OPERATIONS
Timers_Driver.Create; Timers_Driver.Start;Sensors.Start;Alarm.Start;Bargraphs.Init;
PROPAGATED_EXCEPTIONSNONE
HANDLED_EXCEPTIONSNONE
PSEUDO_CODE
TBD
CODE
TBD
END_OPERATION Start
OPERATION Stop
DESCRIPTION
This operation stops all the system (the alarm, the bargraphs, the sensors) and resets the monitoring timer.
USED_OPERATIONS
Timers_Driver.Delete;
Alarm.Stop;
Bargraphs.Switch_Off;
Sensors.Stop;
PROPAGATED_EXCEPTIONSNONE
HANDLED_EXCEPTIONSNONE
PSEUDO_CODE
TBD
CODE
TBD
END_OPERATION Stop

OPERATION Is_Value_out_of_Range
DESCRIPTION
TBD
USED_OPERATIONS
TBD
PROPAGATED_EXCEPTIONS
TBD
HANDLED_EXCEPTIONS
TBD
PSEUDO_CODE
TBD
CODE
END_OPERATION Is_Value_out_of_Range

OPERATION Monitor
DESCRIPTION
This operation is in charge of the EMS monitoring. It acquires the mean values of each sensor, controls them against limit values with respect to each category of sensors, converts those mean values into displaying values according to min. and max. information and displays them on the corresponding bargraph. It sets the alarm if the mean value is out of range or if there is a sensor failure. It also set the green colour of a bargraph when the value is correct, the red colour when the value is out of range and the red flashing colour when the corresponding sensor is failed.
The Monitor operation switched the alarm on for each out of range value and each failed sensor and switched the alarm off for each correct value and running sensor.
USED_OPERATIONS
Alarm.Switch_On;
Alarm.Switch_Off;
Bargraphs.Display;
Bargraphs.Flash;
Bargraphs.Set_Colour;
Sensors.Acquire;
Is_Value_out_of_Range;
PROPAGATED_EXCEPTIONS
NONE
HANDLED_EXCEPTIONS
Sensors.SENSOR_FAILURE;
PSEUDO_CODE
TBD
CODE
TBD
END_OPERATION Monitor
END_OBJECT *Ctrl_EMS*

1.5.4 EXAMPLE OF ADA CODE IMPLEMENTATION

In the following we give the Ada code generated from the ODS of object EMS and Ctrl_Ems. *APPENDIX A2* - of this document gives the same code where ODS text parts have been generated within the code as Ada comments.

```
with Ctrl_EMS, Alarm;
package EMS is
-- DESCRIPTION
   procedure Start renames Ctrl_EMS.Start;
   procedure Stop renames Ctrl_EMS.Stop;
   procedure Acknowledge renames Alarm.Acknowledge;
end EMS;
```

Figure 34 - Ada Specification associated to EMS ODS

```
package Ctrl_EMS is
      task OBCS_Ctrl_EMS is
         entry Start;
         entry Stop;
         entry Monitor;
      end OBCS_Ctrl_EMS;
   procedure Start renames OBCS_Ctrl_EMS.Start;
   procedure Stop renames OBCS_Ctrl_EMS.Stop;
   procedure Monitor renames OBCS_Ctrl_EMS.Monitor;
end Ctrl_EMS;
```

Figure 35 - Ada Specification associated to Ctrl_EMS ODS

```
    with  Sensors;
    with  Alarm;
    with  Bargraphs;
    with Timers_Driver;
 package body Ctrl_EMS is
       IT_1HZ_ADDRESS : constant := Monitor'ADDRESS;
       procedure Is_Value_out_of_Range;
       procedure OPCS_Start is separate;
       procedure OPCS_Stop is separate;
       procedure OPCS_Monitor is separate;
       task body OBCS_Ctrl_EMS is
       begin
              loop
                 select
                    accept Start;
                       OPCS_Start;
                 or  accept Stop; -- empties Stop queue
                       OPCS_Stop;
                 or  accept Monitor; -- empties Monitor queue
                       OPCS_Monitor;
                 end select;
```

Figure 36 - Ada Body associated to Ctrl_EMS ODS

```
      procedure OPCS_Start is
         begin
            Timers_Driver.Create (MONITORING_TIMER,
                                  MONITORING_FREQUENCY,
                                  IT_1HZ_ADDRESS );
            Alarm.Start;
            Bargraphs.Init;
            Sensors.Start;
            Timers_Driver.Start ( MONITORING_TIMER );
         end OPCS_Start;

      procedure OPCS_Stop is
         begin
            Timers_Driver.Delete ( MONITORING_TIMER );
            Sensors.Stop;
            Alarm.Stop;
            Bargraphs.Switch_Off;
         end OPCS_Stop;
      procedure Is_Value_out_of_Range is
         begin
            null;
            -- TBD
         end Is_Value_out_of_Range;

      procedure OPCS_Monitor is
         begin
            null;
            -- TBD
         end OPCS_Monitor;
   end Ctrl_EMS;  -- end body of package Ctrl_EMS
```

Figure 37 - Ada Specification associated to EMS ODS (Continued)

2 ADVANCED CONCEPTS

2.1 ARCHITECTURAL GUIDELINES

In this section we shall give some guidelines for identifying HOOD objects. There are however no strong principles, if any, for finding objects straight away for a HOOD design.

The Basic design Step describes how the software requirements are reformulated and how one can go towards an Informal Solution Strategy in informal natural language (e.g. English). From this text, nouns may be identified as candidate objects, and verbs may be identified as corresponding candidate operations. This process is repeated for each object that is decomposed.

The intent of an object is to represent a problem domain entity, either a real world object or a data structure, and the object should act as a black box by hiding the data, and allowing access only through operations - thus allowing easy testing, debugging and maintenance.

The principles of abstraction and information hiding provide the main guidelines for assessing an object; thus a "good object" represents an encapsulation of a problem domain entity and hides internally information that is likely to change if the implementation changes. One can define a scale of abstraction ranging from the real world object to purely logistic objects as follows:

- **Entity abstraction**
  represents a useful problem domain entity. Such an entity would be as concrete as a hardware device or more abstract such as a compiler intermediate file.
- **Action abstraction**
  a child object that represents the state of a parent object can be seen as an abstraction of the actions of the parent object. This may have a State Transition Diagram, and may be Active or Passive according to the type of the parent object.
- **Logistic abstraction**
  an object that represents a data store that is used within the design, without an external interface is a logistic abstraction.
  Examples are stack, list and tables.

At the top level of the design, in decomposing the root object, candidate objects are generally real world, problem domain objects (e.g. hardware devices, input devices that drive system activity, output devices that respond to system activity), or are data (files as collections of input or output data), either transient data such as messages or commands, or data stores, such as stacks, lists or tables,etc ....
At lower levels of design decomposition, further general types of object may appear such as states, data pools, data records,..... .

In the early time of HOOD, there has been numerous attempts to define *what is a good object*, and we shall not try to do it again. Rather we shall recommend to define HOOD objects along two main principles:

- **identification of root objects from conceptual abstractions** and HOOD parent-child refinement to reduce complexity
- **identification of abstract data type** implementations.

Enforcing these *two guidelines in object identification is a good start*, at our experience, to come to "good solutions"[1]:

- the identification of objects based on conceptual modelling and abstraction allows to catch the *good structure*, i.e. one that is possibly invariant from one application to another in a given domain. (thus reusable for a similar project)
- the identification of *abstract data type implementations*, is also a way to achieve good logical grouping of data and related operation (with strong logical cohesion thus enforcing software engineering principles) and going as well towards full object oriented structure: that is systems structured with only ADTs[2]..

Furthermore, we stress here again that HOOD emphasizes structuring through modules whose interfaces can be mastered, subcontracted, rather than on inheritance structures which expresses implementation factorising. At the worst (i.e. when two ADTs objects inherits from a third one, and are allocated to different "**work package** modules"), it means that **allocation work should be reconsidered, not the design!**

Hence the importance of a phased approach in the development process, (see section 2.1.2 below) where **logical, conceptual design solutions are produced first, and physical, implementation solutions, including work-package allocation are defined second.**

2.1.1 STRUCTURING BASED ON ADTS

The definition of abstract data types is based on the principle of extending and building complex types from primitives ones, directly supported by the target language. This technique allows a designer to structure and enforce "type properties" on complex data handled in a solution.

In HOOD an implementation of an abstract data type is defined through an object that encapsulate the definition of the type and associated operations. Such objects are just normal HOOD objects encapsulating all what is related to an abstract data or process type implementation, and in order to distinguish them from non abstract data type support ones [3]we shall in the following refer to such objects as HADT(**H**OOD **A**bstract **D**ata **T**ype) objects.

2.1.1.1 Object Abstraction

There are two ways for capturing abstractions: Abstract Object (ADO) and Abstract Data Types (ADT). ADO allows to define one (and only one) object at a time. This object has an internal state which is represented by internal data. This data is not directly accessible by the user of the object (it is hidden as stated in the previous section), but it can be manipulated in a correct way thanks to the provided operations.
Note that HOOD graphical view only shows the provided operations (see figure 38), the rest of the provided interface (types, constants and exceptions) are only appearing in the textual ODS (see figure 39).

[1]. may not the best ones, but good ones.
[2]. (whether these ADTS belong to an inheritance hierarchy is rather a detailed design implementation aspect)
[3]. sorry not everything in a system can be modelled as an abstract data or process type implementation.

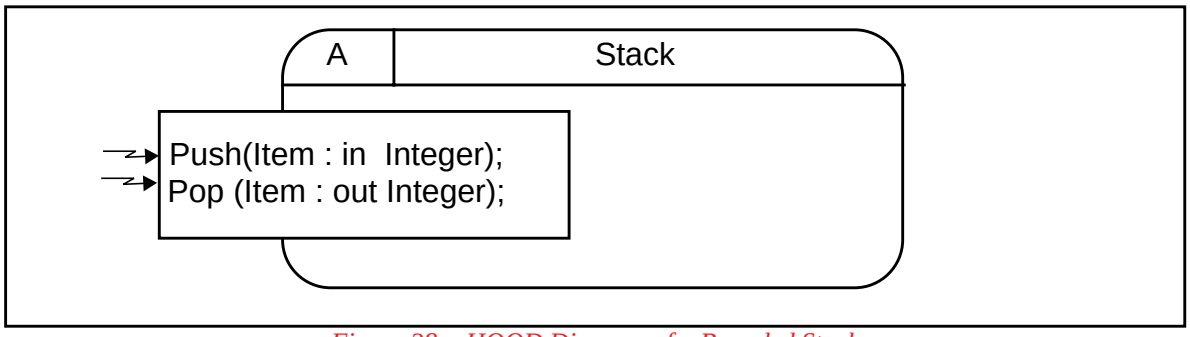


Figure 38 - HOOD Diagram of a Bounded Stack

In order to illustrate the importance of the provided interface, which may give a lot of indirect information, let us try to understand what implicit assumptions may be contained in the provided interface of an example (see figure 39).

First, the presence of a provided constant indicates a static memory allocation scheme. The existence of the exception `X_OverFlow` tends to corroborate this hypothesis.

```
OBJECT STACK is ACTIVE
   DESCRIPTION
      A protected integer stack as an abstract Data Object
   IMPLEMENTATION_CONSTRAINTS
   --| constraints |--
   PROVIDED_INTERFACE
      CONSTANTS
         MAX_STACK_SIZE : constant := 100 ;
      OPERATIONS
         Push(Item : in  Integer) ;
         Pop (Item : out Integer) ;
      EXCEPTIONS
         X_OverFlow  raised_by Push;
         X_UnderFlow raised_by Pop;
   OBJECT_CONTROL_STRUCTURE
      DESCRIPTION
         pop and push are simple constrained operations
      CONSTRAINED_OPERATIONS
         Push CONSTRAINED by HSER;
         Pop CONSTRAINED by HSER;
   INTERNALS
      TYPES
         Stack_Range is Integer range 1 .. MAX_STACK_SIZE;
         Stack       is array ( Stack_Range) of Items;
      DATA
         Top       : Integer := 0 ;
         The_Stack : Stack;
. . .
```

Figure 39 - Textual view of a Bounded Stack defined as an ADO

The two provided exceptions seem to be very similar in the sense that they are raised when the use of the stack leads to violate its boundaries. But they differ fundamentally. An abstract stack is infinite and the `X_OverFlow` exception is raised when the concrete stack is not big enough to simulate the abstract one, what is an implementation limit. The `X_UnderFlow` exception is raised when the user tries to `pop` an empty stack, which is clearly an error in the implemented algorithm. This exception can be useful during the debugging phase but has no more interest in the final system where this error is not likely to appear.

Anyway, it could be still better not to use exceptions, except for predefined exceptions.
In the above example, a control parameter could be used for `pop` and `push` operations as a return parameter. This is a typical example of avoiding using exceptions.
An alternative to the bounded stack with no exception is given in figure 40.

```
OBJECT UNBOUNDED_STACK is ACTIVE
DESCRIPTION
   The stack is now unbounded, that is to say the constant MAX_STACK_SIZE disappears, as well
   as the X_OverFlow exception |--
PROVIDED_INTERFACE
   CONSTANTS
      none
   OPERATIONS
      Push(Item : in  Integer) ;
      Pop (Item : out Integer) ;
      is_Empty   return Boolean ;
      is_on_the_Top return Integer ;
   EXCEPTIONS
      none
   OBJECT_CONTROL_STRUCTURE
      DESCRIPTION
   Pop is constrained and protected by a guard condition so that it blocks a client when this one tries
   to pop an empty stack
...
```

Figure 40 - An alternative to the previous stack

The next point concerns the provided operations: as it is defined, the interface is very small and in particular there is no “access function”, i.e. there is no way to ask information about the state of the object (“is the stack empty?”, “is the stack full?”, “what is the value on the top of the stack?”,...). Such kind of operations is important for the reusability of the object but is less useful in a concurrent context[4].
The last point to mention is the fact that operations are constrained. It can be in contradiction with the exception declaration: `pop` and `push` may be guarded by blocking conditions so that the X_OverFlow and X_UnderFlow situations of the stack will never happen.

In any case, we have shown that the only formal information contained in the provided interface is subject to multiple interpretations, the informal fields must be used to clarify the designer intentions.

2.1.1.2 Abstract Data Type Abstraction

Defining an Abstract Data Type (ADT) is very similar to Object Abstraction as presented above but, instead of designing a single object, a general model of similar objects is defined. In fact, at least in its passive form, an ADO can be seen as an instance of an anonymous ADT. Instead of only providing operations for manipulating internal hidden data, a private type is also provided with the same set of operations. Each provided operation has an additional parameter describing which object is manipulated. Internally, data is not directly declared, but the private type is precisely defined. This type will be used by clients to declare data of this type in their own context (see figure 41). Implementation issues discussing the localization of data instances is further discussed in section 2.1.1.3 below.

[4] imagine you want to pop the stack, you verify before proceeding that it is not empty but nobody can insure that another task has not emptied the stack in between the call of `is_empty` and the call of `pop`.

.The real “objects” derived from this definition are data values of the defined ADT. They are not graphically represented on the HOOD diagrams. But they will appear as part (internal data) of the terminal HOOD objects using this type.
As a matter of fact, objects encapsulating ADT are useful for defining the low-level objects of the problem, those which are likely to be found at several places in the decomposition tree, and those sufficiently general to be stored in reuse-libraries.

```
OBJECT STACKS is PASSIVE
   DESCRIPTION
      A simple integer stack as an abstract data type |--
   IMPLEMENTATION_CONSTRAINTS
      --| constraints |--
   PROVIDED_INTERFACE
      TYPES
         T_Stack is limited private ;
      CONSTANTS
         MAX_STACK_SIZE : constant := 100 ;
      OPERATIONS
         Push(Stack : in out T_Stack; Item : in  Integer) ;
         Pop (Stack : in out T_Stack; Item : out Integer) ;
      EXCEPTIONS
         X_OverFlow ;
         X_UnderFlow ;
   INTERNALS
      TYPES
         Stack_Range is Integer range 1 .. MAX_STACK_SIZE;
         Stack_Body  is array ( Stack_Range) of Items;
         T_Stack  is record
            Top  : Integer := 0;
            Body : Stack_Body;
         end record;
   ...
```

Figure 41 - A bounded stack defined as an ADT

2.1.1.3 ADT implementations as HADT objects

Two kind of implementations may be developed and are illustrated below:

- *Implementation encapsulating data instances (Values of the type).* This is where the provided interface of the module groups all operations manipulating the type, including a create operation. Figure 42 -and Figure 43 - illustrate such a module for STACK. The provided type T_STACK is not a type able to store full values of that type, but simply an identifier of such value (in our case a stack object). Hence clients of such a module can require this type, and may only instantiate identifier values on the type. Full type instances are then produced by executing a STACK.create operation.
- *Implementation encapsulating types only.* Such an implementation is directly supported by a number of target languages (especially Ada) and does not provide a create operation. The associated module rather provides a type defining a data structure able to store values of the type. Hence clients of such a module can require the type, and instantiate full data instances. Figure 44 - and Figure 45 - illustrate such a HADT implementation.

The impacts of these implementation choices must be clearly expressed:

- *encapsulating data instances* allows a strong grouping, easing test and integration with non object oriented programming targets and with distributed development teams.
- *non encapsulating data instances* leads to a grouping of common code to manipulate values

of same type*: it is an implementation of the object oriented class concep*t. **This implementation is the preferred one especially when the target language is an object oriented one like C++**.

Since data instances (values of the type) are separated from the execution operation code, **this implementation must also be used each time the target is to be distributed across several memory partitions, or is planned to evolve towards a distributed one**.

When the type can furthermore be parameterized by other more primitive types, one can define a HOOD class object (or GENERIC), whose instantiation will provide the final type.

2.1.1.3.a HADT Object encapsulating DATA INSTANCES

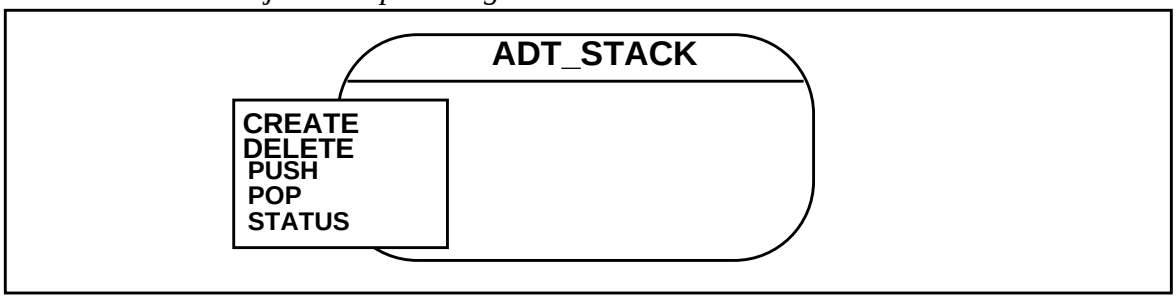


Figure 42 - Graphical representation of object ADT_STACK

The description skeleton of a HOOD object implementing the abstract data type stack could be the following:

```
OBJECT ADT_STACK IS PASSIVE
DESCRIPTION
   Implementation of an abstract Data Type STACK and provides all operations to manipulate values of that type,
   creation included.
IMPLEMENTATION CONSTRAINTS
   Dynamic heap memory is limited to 1 MB
PROVIDED_INTERFACE
   TYPES
      T_Status is (BUSY, IDLE, UNDEFINED);-- definition of a type
      T_STACK; -- the type structure is hidden to clients
   OPERATIONS
      CREATE (STACK: out T_STACK; Size : in integer); --constructor
      PUSH (STACK: out T_STACK; Element: in T_Element);
      POP(STACK: in T_STACK; Element: out T_Element);
      STATUS (STACK: in T_STACK; Status: out T_Status);
      DELETE (STACK: in T_STACK;); --destructor
   EXCEPTIONS
   X_OUT_OF_MEMORY;-- raised_by create when memory limits
REQUIRED_INTERFACE
   OBJECT ADT_ELEMENT
   TYPES
   T_Element; -- ADT_STACK requires type T_Element, provided by the HADT object ADT_ELEMENT.
INTERNALS -- hidden part of the object (not described here)
END_OBJECT ADT_STACK
```

Figure 43 - Structure and ODS of HOOD object ADT_STACK encapsulating Data Instances

The declaration of a STACK object in the DATA field of a client ODS would then be the following:

```
INTERNALS
   DATA
   STACK1: ADT_STACK.T_STACK; -- creates a STACK Identifier
   --initialisation is done by calling ADT_STACK.CREATE(STACK1);
   --all values of STACK1, will be located within the ADT_STACK memory_space.
```

Note: the implementation of HADT STACK groups both execution code and STACK values.

2.1.1.3.b HADT Object encapsulating no DATA INSTANCES

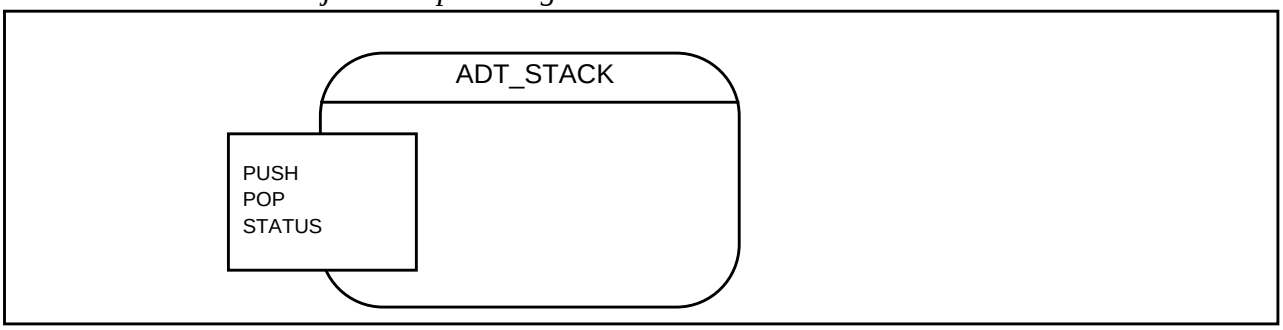


Figure 44 - Graphical representation of object ADT_STACK

The description skeleton of a HOOD object implementing the abstract data type stack could be the following:

```
OBJECT ADT_STACK IS PASSIVE
DESCRIPTION
   Implementation of an abstract Data Type STACK and provides all operations to manipulate values of that type,
   creation included.
IMPLEMENTATION CONSTRAINTS
   Dynamic heap memory is limited to 1 MB
PROVIDED_INTERFACE
   TYPES
      T_Status is (BUSY, IDLE, UNDEFINED);-- definition of a type
      T_STACK; -- the type structure is hidden
      is array (<>) of T_ELEMENT; -- or visible to clients
   OPERATIONS
   PUSH (STACK: out T_STACK; Element: in T_Element);
   POP(STACK: in T_STACK; Element: out T_Element);
   STATUS (STACK: in T_STACK; Status: out T_Status);
   EXCEPTIONS
      none;
REQUIRED_INTERFACE
   OBJECT ADT_ELEMENT
   TYPES
   T_Element; -- requires type T_Element, provided by the HADT object ADT_ELEMENT.

INTERNALS -- hidden part of the object (not described here)
END_OBJECT ADT_STACK
```

Figure 45 - Structure and ODS of object ADT_STACK

The declaration of a STACK object in the DATA field of a client ODS would then be the following:

```
INTERNALS
   DATA
   STACK1: ADT_STACK.T_STACK(200);
   -- creates a STACK of 200 elements
   -- STACK value will stored in the associated data structure within client's memory space
.
```

Note: an implementation of the above ADT_STACK object groups common execution code all type instances: it is an implementation of the object oriented class concept, grouping code and class instances only, whereas object/data instances are declared in client module memory-space.

2.1.1.4 Defining Logical Interfaces with ADT support

The use and definition of HADT object allows to define an object as the interface between two others which exchange complex data.

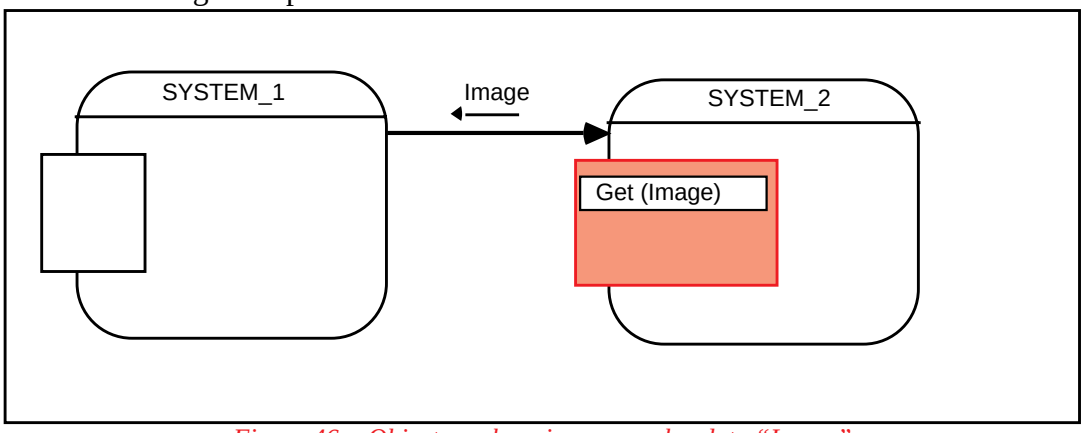


Figure 46 - Objects exchanging a complex data "Image"

The addition of this object can be made without any change in the object provided interface[5] of the two others, and thus **is a technique of refinement of an initial HOOD model.** Figure 47 - below shows that the object 'ADT_Image' is the interface object associated to the exchanges of image data between SYSTEM_1 and SYSTEM_2 objects
The object 'ADT_Image' can be defined as an implementation of an abstract data type "image" and as environment object for the current hierarchy dealing with SYSTEM_1 and SYSTEM_2 objects.

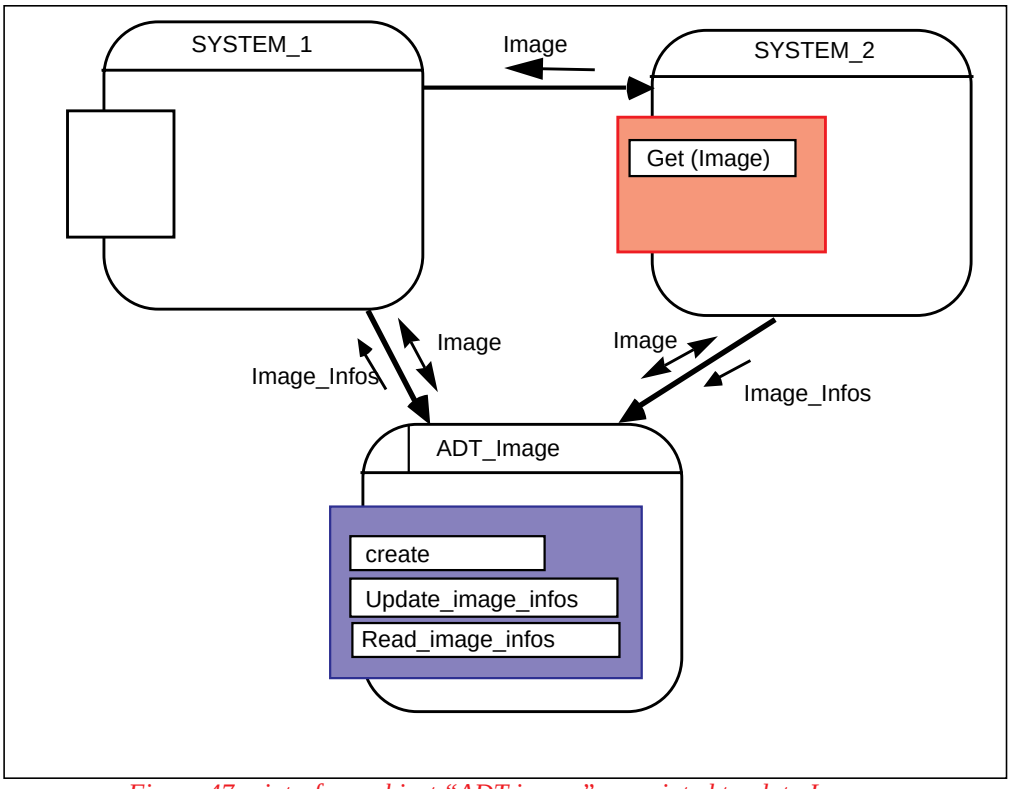


Figure 47 - interface_object "ADT image" associated to data Image.

[5] however changes have to be made in the Required Interfaces

2.1.1.5 ADT Refinement Techniques

Associating additional HADT objects to formalize the exchange of data between two objects of initial model is thus a technique of refinement, **leaving the initial model invariant** from the point of view of its provided interfaces. Thus :

- logical initial architectural models can be expressed, and refined according to specific project and target constraints,
- formal definition of interfaces can be achieved at a high level of abstraction.

Furthermore, the intensive use of this refinement technique allows to serialize the problems attached to an implementation. Separate descriptions can be attached to different level of abstractions, and the definition of libraries of reusable objects (defining a high level abstract interface and different implementations for different targets) is supported.

2.1.1.6 Deriving HADT objects from DataFlows

Figure 48 - Figure 49 -and illustrate how a complex data (*message*) may be defined as a data instance of an abstract data type. The associated type MESSAGE is either:

- provided by the BUFFER object or
- provided by one of its brother object. However such a solution would introduced additional visibility relationships, achieving a less structured and maintainable code.
- provided by another HOOD environment object.
- provided by a HADT object defined as an object of the current hierarchy
- provided by a HADT object defined as an environment HADT, fully reusable from other hierarchies and/or designs.

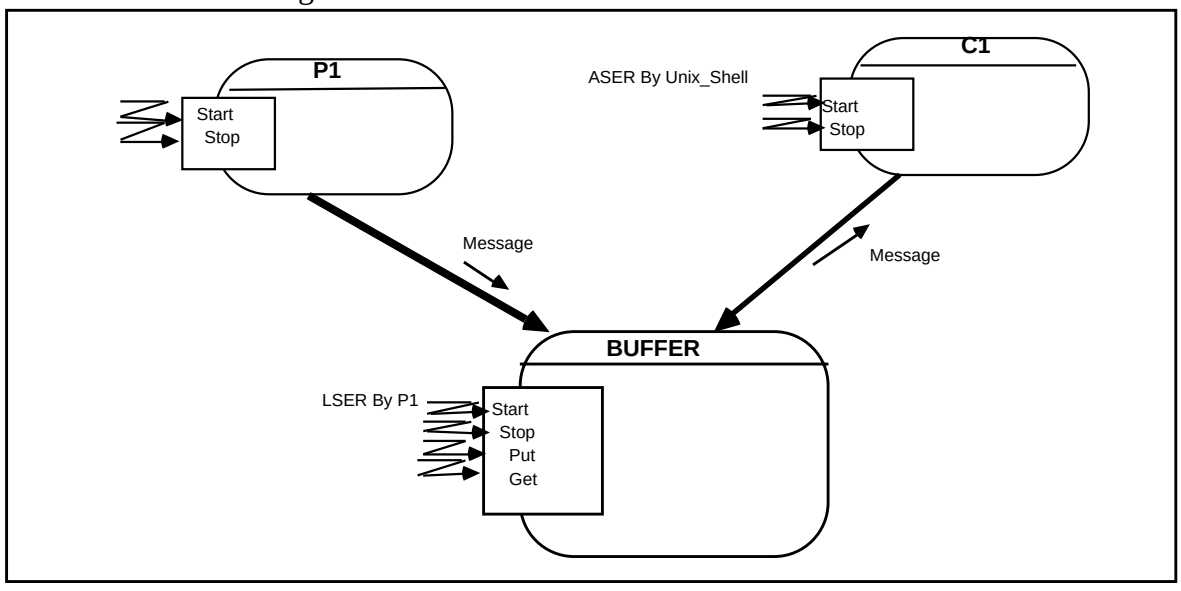


Figure 48 - Objects of initial HOOD model exchanging a complex data "Message"

It is clear following software engineering principles that we shall favour the last HADT object solution that defines a reusable module of high logical cohesion dealing only with message handling operations/code (see Figure 50 -).

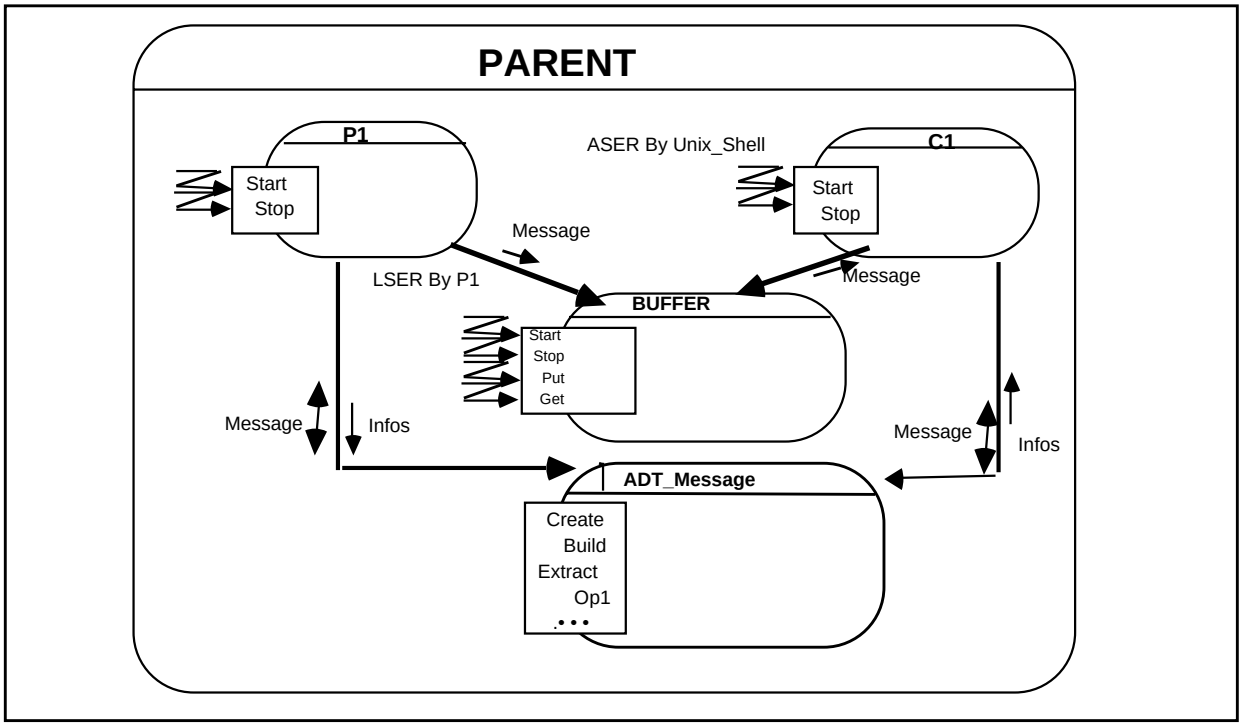


Figure 49 - HADT object ADT_MESSAGE providing a MESSAGE class

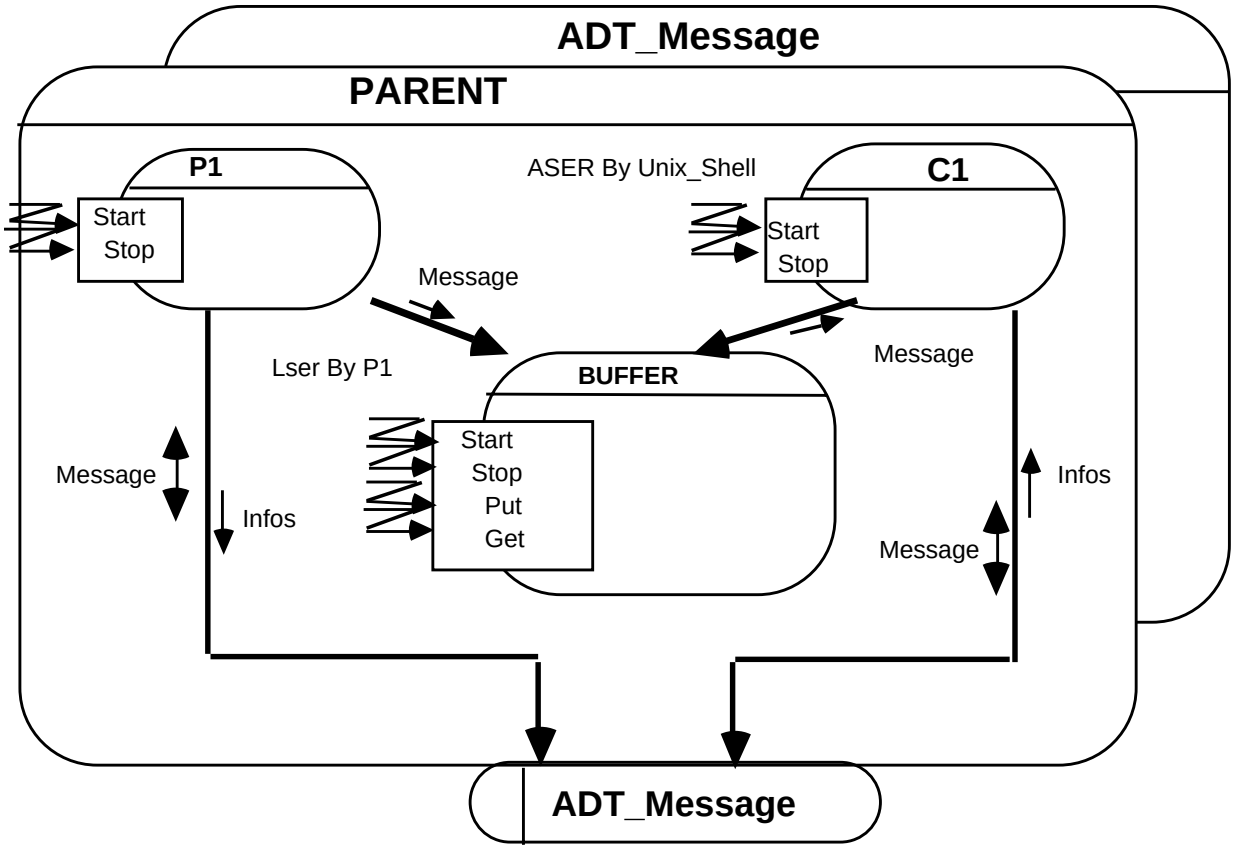


Figure 50 - ADT object ADT_MESSAGE providing a MESSAGE class

2.1.2 THE HOOD DESIGN PROCESS AS SEVEN DESIGN RULES

The HOOD decomposition approach may eventually be summarized into seven design rules, where the system to design is first represented as an object related to its environment, and then *broken down into objects according to three refinement lines* as represented in Figure 51 - below :

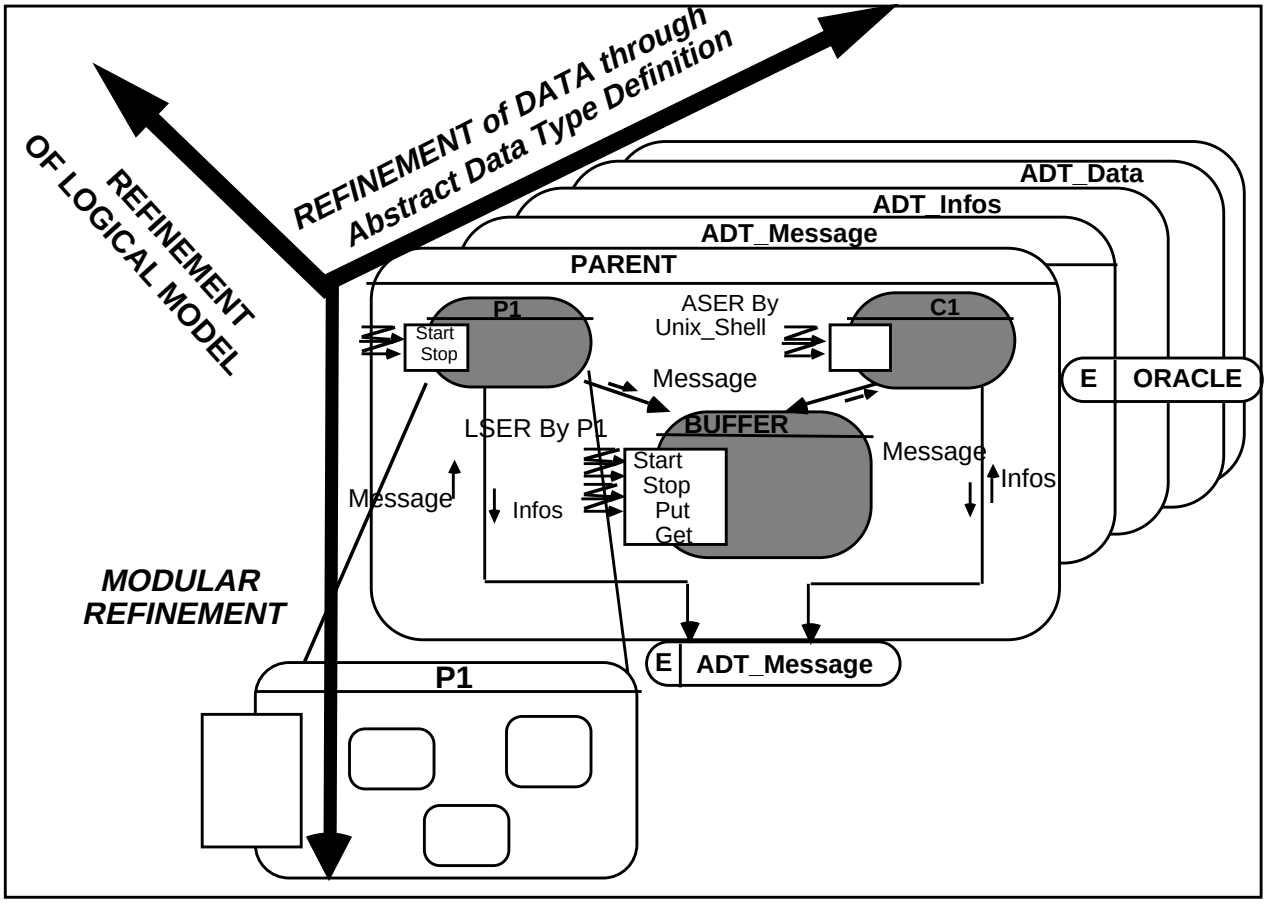


Figure 51 - HOOD Method of decomposition and refinement[6]

- *modular decomposition line:* this refinement line tries to find a modular structure in terms of object/component exchanging data. For each level of parent/child decomposition, one applies standard decomposition criteria based on allocation of functions to objects and that define loosely coupled objects, with minimised provided interfaces.
- *abstract data type refinement line:* this refinement line tries to specify groupings of data manipulation operations, into associated HADT supporting objects. Each dataflow identified in previous line can be implemented as an instance of an abstract data type (or a basic type of the target language). The operations on the data are identified as the client objects are further refined, thus this refinement is performed in parallel with the modular refinement. When an HADT object provided interface is fully defined, it can, in turn, be designed following a modular decomposition refinement and ADT refinement.
- *logical to physical refinement line*: this refinement line is not really a refinement, but rather a "restructuring" of the logical objects into "groupings of objects" that fit and map into target constraints. It corresponds to the distribution phase identified described in section 1.3.5 above. For example, a grouping of the logical objects is needed when the target system is

[6]. this figure shows the process for the decomposition of one root object, but for large projects, this process may be performed in parallel on several root objects.

distributed over two processors (or heavyweight processes/tasks[3]). In order to avoid network communication bottlenecks, objects will be allocated to one or the other processor according to the functions they support, and at the same time trying to avoid heavy remote induced dataflow. This grouping can be formally stated through the definition of a physical model in terms of HOOD virtual nodes, matching the target processors, and then allocating objects from the logical model to these VNS. Hence the designer gains advantages of automated code generation, that have as principle not to modify the logical code of allocated objects.

According to the principles summarised above, on can identify **seven rules to find the objects**, which are as follows :

a **Start defining the system-to-design** as an *interface* (set of provided and required services) to its environment (possibly modelled as a partition of environment objects).

b **Define the key objects** abstracting invariants of the system. Such definition may be done according to :

c functions allocated to the child objects of the system-to-design

d identification of associated dataflows between these objects (note that dataflows are related to allocated functions)

e **Define the implementation of the communication and dataflows** between these objects and environment objects.

f **For each data previously identified, define associated abstract data** type as HADT objects (unless the data can be directly defined as an instance of a type of the target language). It may happen, that operations that manipulate the data of the type are still vague (create, delete, update), but this is not a problem at this stage. When the refinement of objects identified in steps 2) and 3) is performed, new operations on the data will be found, and the provided interface of the HADT object will then be updated in parallel.[7].

g **Resume the refinement of objects identified in 2) and 3)** following the same principles until they can be directly coded, and always trying to identify as much HADT objects as possible.

h **Refine each HADT object** (unless it is already terminal), by applying step 2) and 3) and again identify new objects exchanging data, and possibly new HADT ones.

i **Document target language ODS fields**, or resume HADT refinement using the specifics of the target language

Since HADT objects can be directly implemented or refined into OOP target languages (see section 2.12 below for details), the HOOD development approach allows thus to combine both a design representation target towards flat type structure and object oriented class structure. Whereas several methods for identification of classes are based on analysis techniques that were mainly derived from the Entity-Relationship model extended with inheritance, the HOOD design approach leads naturally to the identification of classes from the definition of logical interfaces and applicative abstract data types. The resulting structure reflects the top down design trade off partition of the software, rather than a bottom up approach derived from implementation data structures.

Taking into account both the natural client-server relationship between objects and classes, orthogonal to the composition relationships, this approach proves to be a powerful structuring tool.

7. Note that the refinement takes place on several hierarchies at the same time! This is one of the most noticeable new feature of HOOD3.1 over the earlier definition HOOD3.0.

It appears today for us, as the only viable integration support for both modular and full object oriented approaches Furthermore, by integrating both abstract data typing and process concepts HOOD3.1 is the software engineering tool of choice, supporting the transition from classical development practice into full object oriented one.

2.1.3 OTHER GUIDELINES FOR IDENTIFYING OBJECTS

Abstraction techniques are the basics for defining objects; however as designer's experience goes larger, other design patterns can be used for the early definition of a system architecture: the top-level architecture is defined as a partition of HOOD objects which may abstract:

- a layered architecture model, possibly hierarchically structured or
- technological development criteria.

Furthermore, the refinement of the top-levels components are constrained by mapping into communication and implementation models.

2.1.3.1 Structuring based on layered models

Every software can be seen as an implementation of a layered model in which the application part are on the upper levels, whereas general purpose services and execution infrastructure services (OS services, communication services) are located at the lower level of the model.

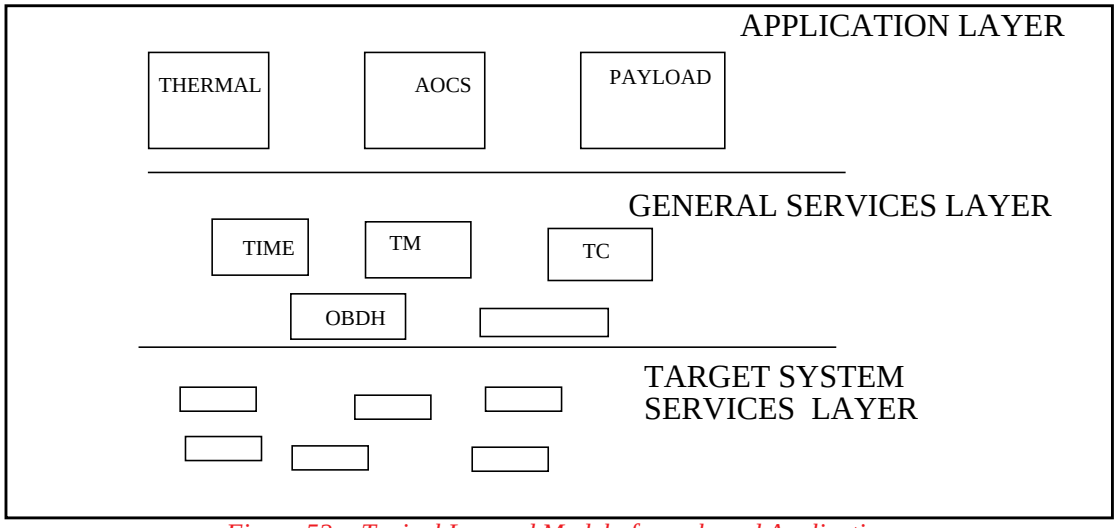


Figure 52 - Typical Layered Model of a on board Application

2.1.3.2 Structuring based on Technological Components

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

The partitioning of a software can thus be made according to development and technological criteria. As a result a general architecture model of information systems can be defined at the top-level comprising:

- a dedicated application part
- an MMI (Man Machine Interface) part
- a DataBase or Storage interface part
- possibly a rule system interface part

Figure 53 - gives an example of a technology component decomposition defined for a large information system.

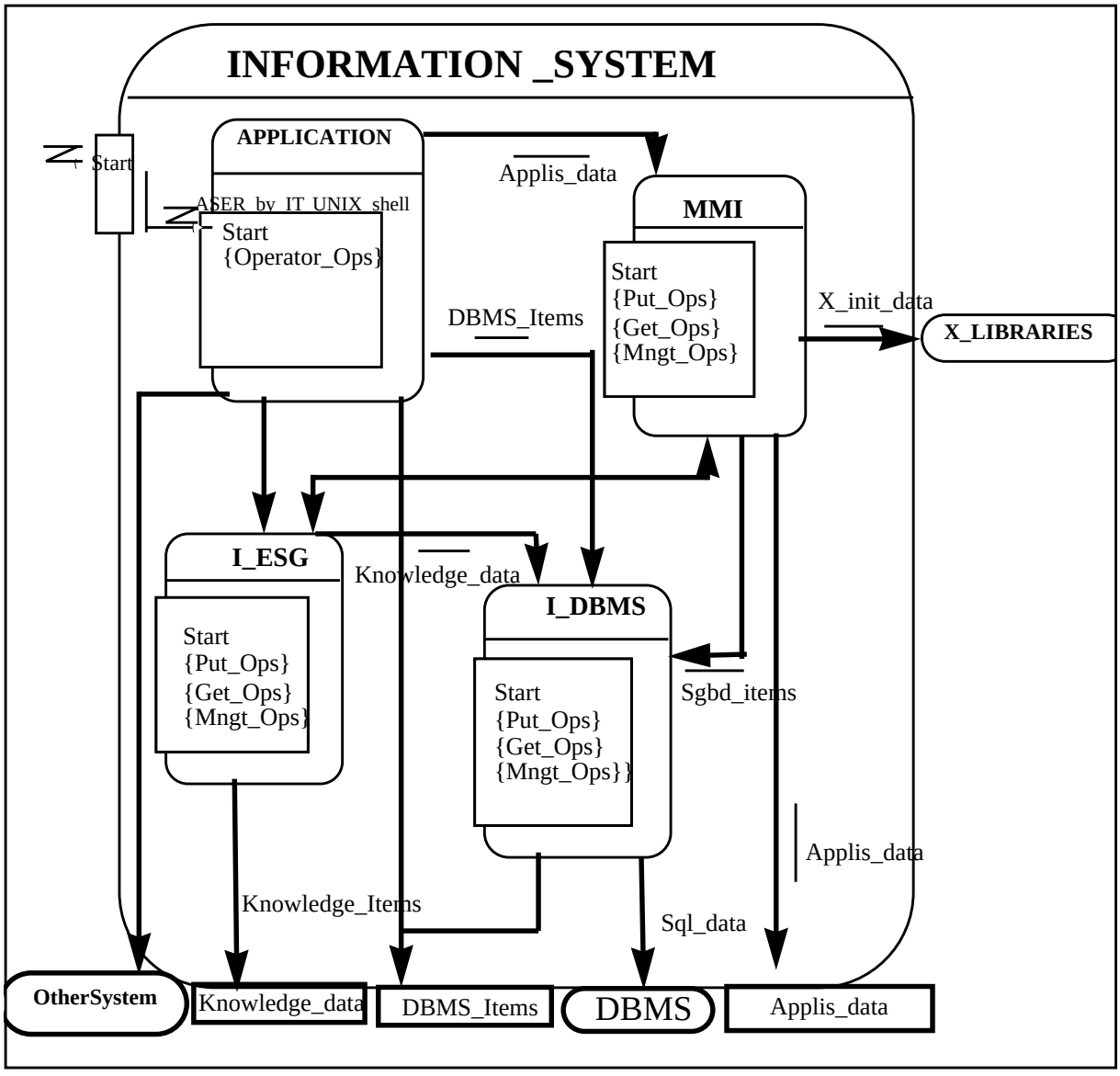


Figure 53 - Typical System Information Model partitioned through Technological Components

Four objects are associated to four development lines and partition the system into technology component objects that exchange data. The latter are again described through HADT objects represented as uncle objects and where:

- the access interface to a datum (with all services provided to its clients) is formalised by the

set of operations that manipulated the associated type(creation, update of a sub field, read, delete, checks, etc....) in a HADT object,

- further refinement of those HADT may produce objects/classes, HADTs of less complexity up to terminal specification directly rewritable in a target (possibly OO) language, is based on the graphical extension and refinement rules(see below).

2.1.3.3 Structuring and Refinement

Once a top-level decomposition has been chosen, it is also necessary, for the architecture to be detailed efficiently, to have a technique for refining the associated interfaces. By refining the interfaces according to standard communication and implementation models, the designer can achieve:

- the definition of reusable components with well established interfaces,
- the definition of generic architecture,
- the definition of application domain specific frameworks, possibly evolving towards standard ones.

The interfaces between the different subsystems or software layers associated to the development activity lines can moreover be formalised as HADT objects that factor the exchanged data and provide for a common representation whatever the activity and technology context.

The above concepts have been synthesized after feedback and trials on numerous projects, built into principles and refinement techniques for integrating multiple technology developments. These principles must be applied all together within a development and list as:

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

2.1.3.4 Modular and ADT Refinement Principles

Break down rules associated to the HOOD include relationship have as principle to hold the properties of the parent object with the ones of the child objects. *Figure 54* - illustrate this modular refinement technique of a given parent object into child objects that exchange data.

2.1.3.4.a ADTs Identification and Interface Refinement

Once components and their data exchanges have been defined at one level, their interfaces can be formally expressed through ADTs (see *Figure 55* -):

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

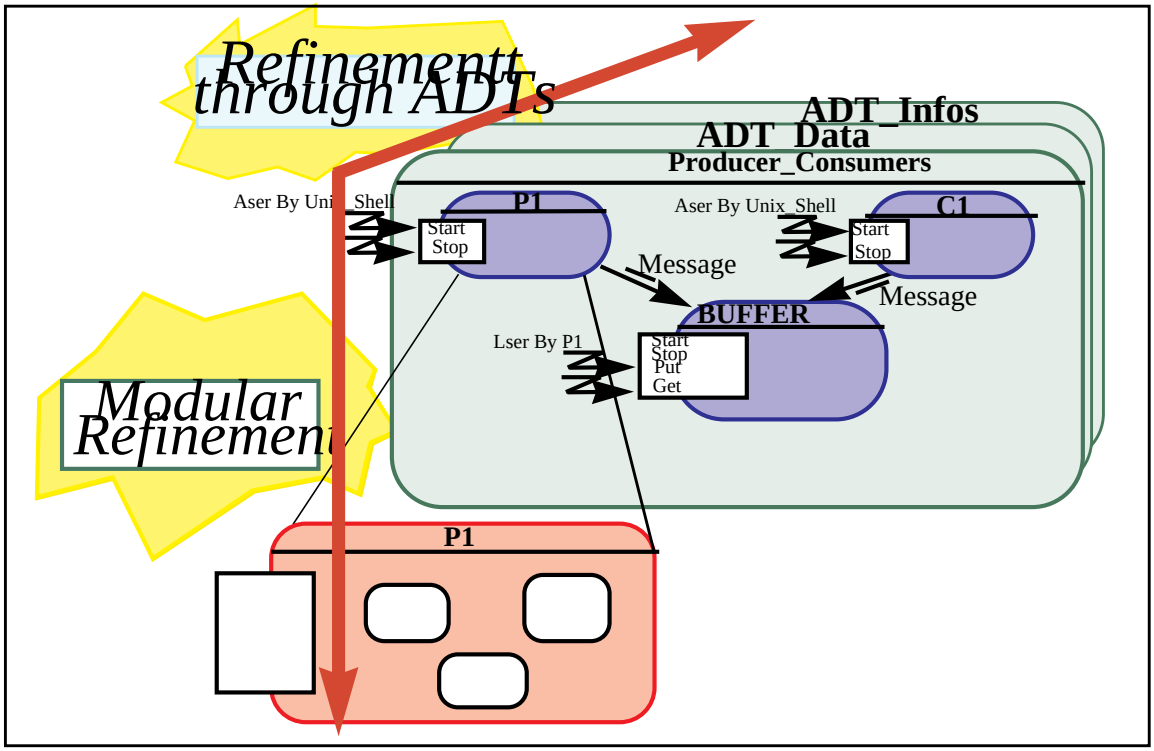


Figure 54 - Modular and ADT Refinement

.

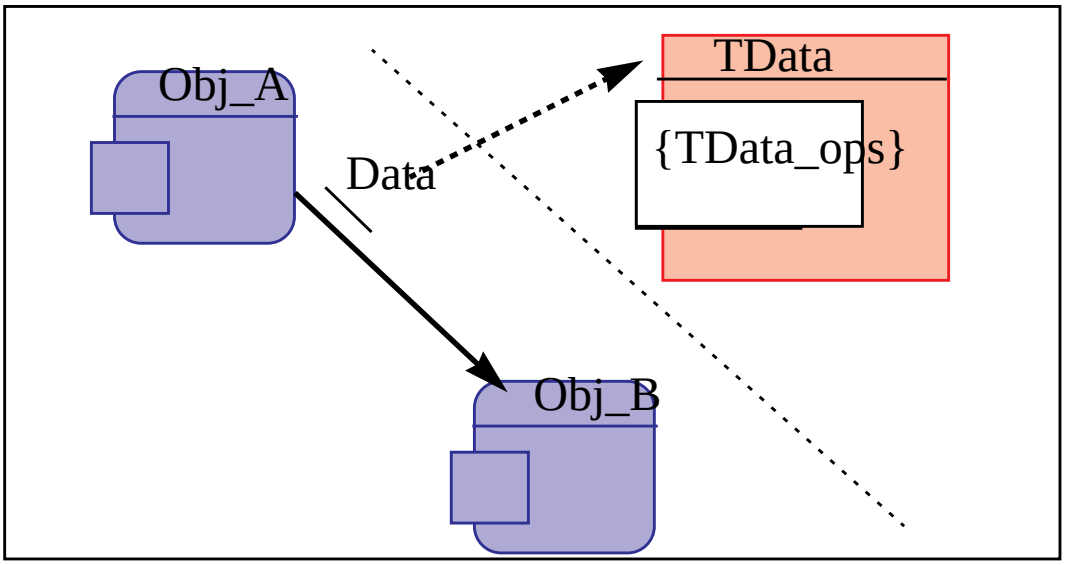


Figure 55 - Principle of specifying Interfaces through ADTs

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

- either as an instance of a primitive type directly supported in the target language, or
- as an instance of an ADT (or a basic type of the target language). The operations on the data are identified as the client objects are further refined, in parallel with the modular refinement. When an ADT provided interface is fully defined, it can, in turn, be refined following a modular decomposition and/or be directly coded in a target language class. Figure 54 - below summarizes these principles.

An ADT implementation is defined as an encapsulation in a HOOD object of operation working on data of that type[8]. For best identification of the type on which the operation works, the receiver of the operation is indicated by the reserver parameter *me*, allowing to distinguish the main type from other parameters.
The difference between an ADT implementation and a target language class are the following:

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

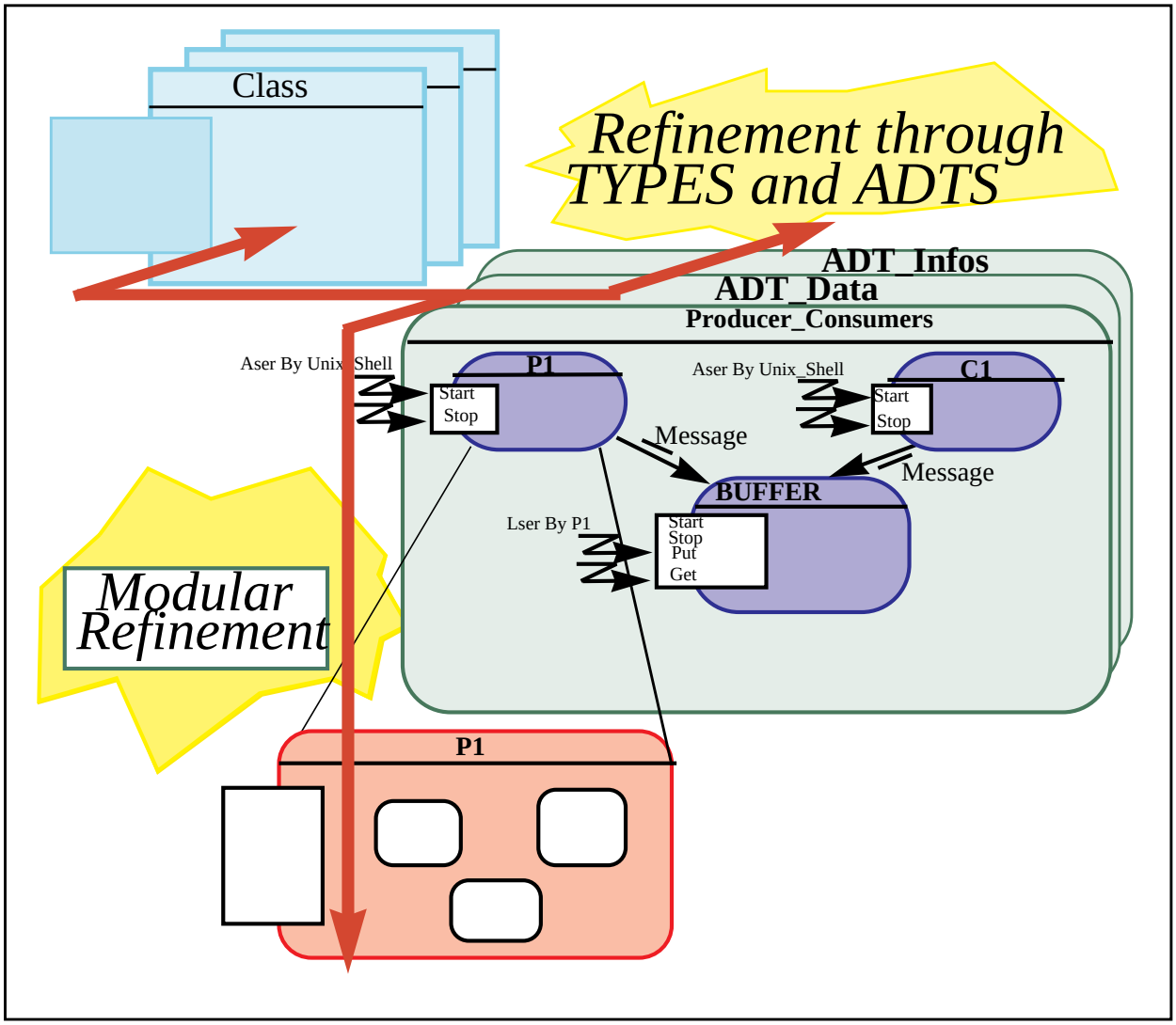


Figure 56 - Combining Modular with ADT Refinement

[8] we mean the ADT

2.1.3.4.b Target Language Class Refinement through Attribution and Inheritance

In the detailed design or coding of HOOD object implementing an HADT three cases may show up:

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

When a target language class is so identified, it is only defined as a HOOD type and through its provided operations, and when the detail design is performed the designer may:

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

2.1.3.4.c Implementation Refinement

This kind of refinement is as the modular refinement; we stress it here only to recall that the designer may elaborate a logical solution that relies on software layers that isolate the system-to-design from the specificities of the target infrastructure and OS. This kind of refinement is illustrated in *Figure 57 -*, is one way to take into account the non-functional constraints such as:

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

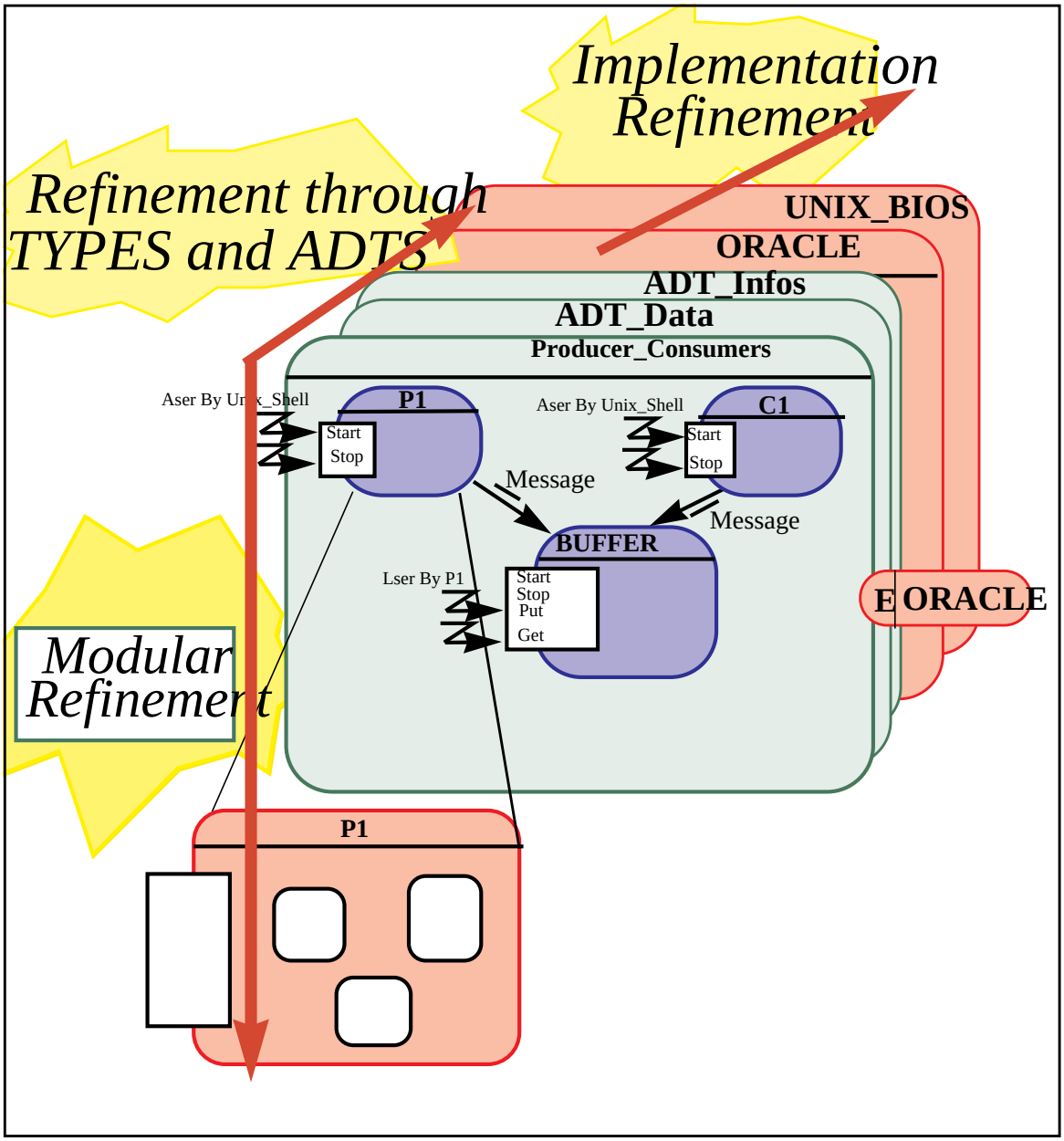


Figure 57 - Refinement Techniques of a HOOD model

2.2 THE HOOD DESIGN DOCUMENTATION

In the following we give a description of *the design documentation* suitable for describing and checking HOOD designs. Be aware that *this is a design documentation description*, not to be confused with a *project documentation* which may need additional documentation items depending on the used documentation standard.

2.2.1 OBJECTIVES

The HOOD design documentation shall favour communication and explication of a solution within a development team. Thus it shall describe the software at different levels of details and abstraction. Also it shall allow QA teams to check that both HOOD approach and description standards have been enforced during a development.

2.2.2 DOCUMENTATION CONCEPTS

HOOD models are represented through a data model where the ODS is the main structuring concept.

The ODS (Object Description Skeleton) is a standard grouping of the object's characteristics organised in structured fields. Informal textual descriptions are defined (*Description, Interfaces, Types, Data, Exceptions and Control Structures as* well as associated semi formal descriptions ***(Operations, Types, Exceptions, Data, Code of OBCS and OPCS).***

Note that the ODS is a logical concept; a physical representation of an ODS is a piece of text grouping the contents of the fields of an ODS into a human readable form. This latter may take different layouts according to the documentation features of the HOOD toolsets and the purpose of the documentation.The associated notations and formalisms can in fact be used for:

- informal verification (through author-reader cycles) of textual descriptions
- design verification (designs checks, pseudo-code)
- code generation for prototyping
- code generation for final products.

The SIF (**S**tandard **I**nterchange **F**ormat) is defined for design exchanges with other HOOD toolsets or development tools. It defines formally the layout in ASCII format of files containing ODS representation in ASCII TEXTs. Hence a SIF representation of an ODS is a valid one, but may not be a very readable document.

2.2.3 DOCUMENTATION MANAGEMENT

The HOOD documentation shall be structured as sets of ODSs, where all or part of the ODS fields are present, depending on the type of document that is produced. For example a design documentation for an architectural document shall group together a set of ODSs, with text sections associated to the basic design step activities included in the DESCRIPTION field of the ODS of the OBJECTS under review. We recommend to structure parent ODS with their DESCRIPTION field structured into H1,H2,and H3 text sections. H3 is a text comprising the textual description of child objects and operations, that can also be taken again as the H1 texts of the ODS associated to the child objects.

A HOOD documentation defines accordingly as:

- a set of ODS, each being unique in the documentation
- a structure that follows:
  - the spaces defined by the object, class and VN hierarchies
  - the parent-child hierarchical relationship

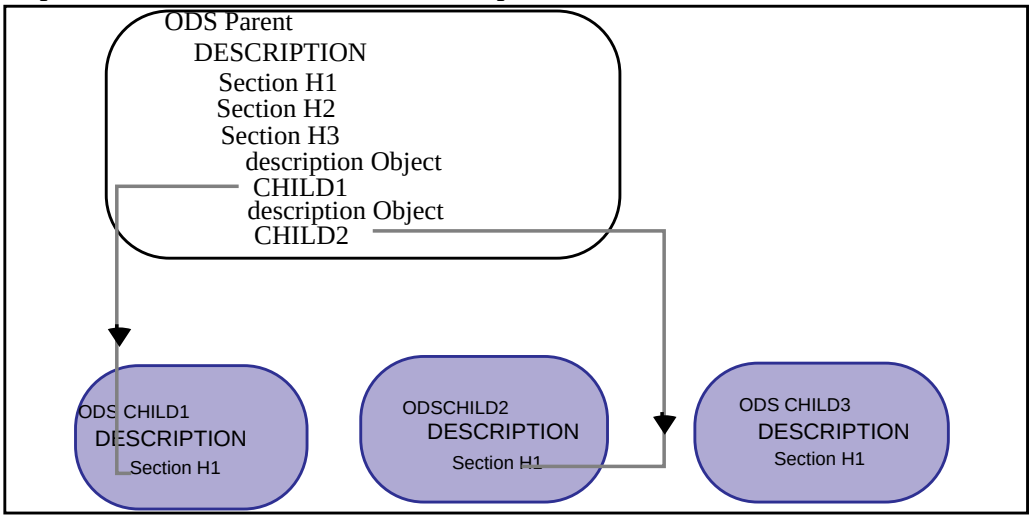


Figure 58 - Relationships between parent and child ODS description sections

2.2.4 DOCUMENTATION ELABORATION

Three states may be distinguished in the ODS life-cycle (see figure below):

- **ODS CHILD:** only ODS fields associated to the interface or user manual are documented.
- **ODS PARENT**: all fields of the ODS are completed. Internal parts only have description of the “implemented_by” relationship between provided resources of the parent and the associated ones of child objects.
- **ODS TERMINAL**: all fields of the ODS are completed and refined in details, especially the fields of the internals that hold the pseudo code and code parts.

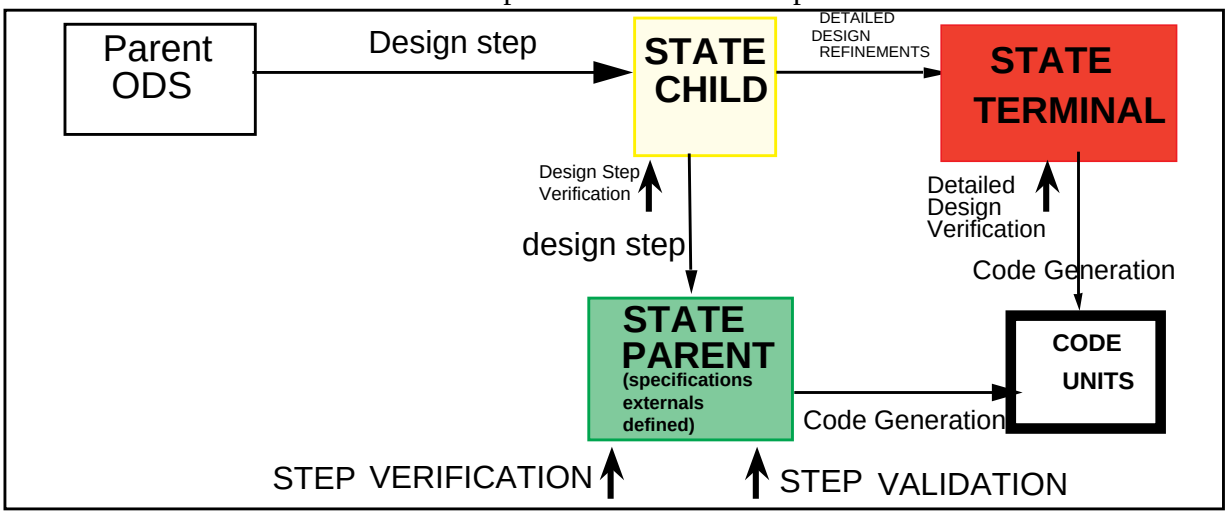


Figure 59 - States in the ODS production life cycle

2.3 EVALUATING A HOOD DESIGN

When the design activities have produced HOOD designs, (taking shape with associated documentation), verification activities shall take place to **ensure that "the design has been produced right"** (and hence is possibly the right design). Those verification activities have been to our experience, the most troublesome issues so far in the HOOD projects. This is because the HOOD design representation may cover several models, several objects, several system configuration, and are possibly constrained to be materialised in enormous linear paper documentation mixing several textual and graphical formalisms.
In the following, we suggest a representation of the HOOD development process as a set of states, where the transition from one state to another is triggered after a verification step.

2.3.1 DEFINITIONS

2.3.1.1 Goal of HOOD design verification

Verification activities take place after design activities and have both technical and organisational goals:

- **technical goals**
  - A HOOD design is a representation of a SYSTEM_TO_DESIGN which starts with a description of a PARENT "root" object with respect to its Environment and is refined into successive descriptions of child objects. In order to ensure consistency of these successive representations up to the detailed design architecture definitions, HOOD defines rules and additional verification procedures may be ensured within each object's scope for each parent-child decomposition step, and from step to step up to terminal objects are reached.
  - Consistency of a HOOD model has to be checked throughout the HOOD process. HOOD recommends reviews at the end of a basic design step, as well as at the end of a level (i.e; when all branches of a tree have been decomposed).
  - Moreover, when a HOOD model is produced, feasibility of the target implementation may to be checked, traceability to requirements should be ensured and the overall quality of the model should be estimated.
- **organisational goals**
  - The work progress must be evaluated in order to help management of a project. The end of design activities (through documentation delivery and/or reviews) at the end of noticeable basic design steps provide candidate milestones for work progress evaluation
  - We shall not take into account here other QA activities here. This is because QA activities' goal is to verify that technical activities have been correctly executed. Depending on projects (where QA procedures are defined and adapted in a QA plan), QA defines procedure that constraint technical verification activities to produce traces of their activity (documentation, verification reports, tests results, etc).And QA procedures check activity by examining these traces.

2.3.1.2 Means for HOOD design verification

Products associated to the design process and activities are of two kinds:

- formal documentation (graphical notations, and/or textual ones that compilable or executable: Ada, Petri nets, Finite State Automata, Esterel etc.)
- informal documentation (texts in natural language)

Evaluation techniques of these products are of two kinds: static verification and dynamic verifications, but are mainly informal techniques, and consists in the following activities:

- inspections (sampling of product or documentation pieces and evaluation with respect to qualitative criteria)
- walk-throughs (assessment of technical contents with error underpinning),
- reader-writer cycles[9] (formal checking of documentation against qualitative and technical criteria with configuration management of errors, remarks, and modifications)
- reviews (checking of product submitted to review with respect to qualitative and technical criteria).
- Automatic or semi-automatic checking can added to these manual means:
- use of design checkers and cross-reference tables that are part of most HOOD toolsets,
- compilation (syntactic checking) and/or execution of fields expressed with a formal notation.

Design validation and verification procedures in HOOD will be based on the separation, in time, of the descriptions of first an external description of an object and then the description of its implementation. Step verification and validation, ensuring consistency of one object decomposition, at the end of a basic design step, is distinguished from level validation ensuring consistency of a complete level decomposition(i.e. several objects together).

2.3.2 DOCUMENTATION FOR VERIFICATION AND REVIEWS

Documents for reviews may be build **by extracting information and texts from ODSs** of concern associated to the model under evaluation at a given date or for a visible point in the project. Some HOOD toolsets allow to build document templates and associated outputs may be produced in PostScript and/or MIF, RTF formats and so, fed in text-processing systems for production of quality documentation according to project documentation standards.(ESA or DOD2167) Documents for specific development activities (Quality control, Unit test Definition, Pseudo code reviews, code reviews,...) may be directly build with a HOOD toolset provided it allows to select relevant ODS and relevant fields for inclusion in a given document.

2.3.2.1 Preliminary Design and Detailed Design Documents

Preliminary design document will only include ODSs in state “PARENT” and “CHILD”.
Detailed design documentation will at least comprise the set of ODS that went from state “CHILD” to state “TERMINAL”. In order to simplify documentation management detailed de-

9.

sign documents may also be defined as preliminary design documents where all ODS of state CHILD" have been replaced by ODS in state "TERMINAL".

2.3.2.2 Documentation for Verifications

Every document taken as input of a development activity should be in a significant state, i.e. contain only the required information for the activity to be conducted. Also the document should be consistent, that is information pieces contained shall be in consistent state one with each other.

- **Documents for checking ODS consistency.** If the object was decomposed, the check will look for consistency between parent descriptions and child implementations. Information items needed are:
  - For a non refined object:
    its ODS in state "CHILD"
  - For a refined object:
    its ODS in state "CHILD".
    its ODS in state "PARENT" if it was decomposed into child objects, otherwise its ODS in state "TERMINAL"
  - The contents of an ODS in state "CHILD" is given in appendix A4.1
  - The contents of an ODS in state "PARENT" is given in appendix A4.2
  - The contents of an ODS in state "TERMINAL" is given in appendix A4.3
- **Documents for Design Step Checking**. The goal here is to check consistency between the child and parent descriptions of the decomposed object and between the parent descriptions and child implementations. Thus the information items needed are:
  - the two ODSs associated to the PARENT, i.e. the ODS in state "CHILD" and the same ODS in state "PARENT"
  - the ODSs in state "CHILD" of the child objects.
  - If the verification addresses a set of N basic design steps, then a set of N such documents would be reviewed.
- **Documents for REVIEWS** The goal here is for reviewers to gain an understanding of the architecture, so that they are confident in the progress of the technical work. T**hus the documentation should target the understanding of the system and provide support for the checking of the object interfaces.** (the level of confidence achieved after the review is such that these are FROZEN and put under change control). Such a documentation may contain:
  - the description of the system_configuration of concern
  - for each system_configuration hierarchy:
  - the description of the design tree
  - for each object of the hierarchy
    Problem definition (section H1.1)
    graphical description of its environment (extracted from section H1.2.2 if the current object is a root, otherwise from H3.4 of the parent of the current object)
    decomposition into child objects and the textual description of how it works, if the current object is non terminal (extracted from sections H2, H3.1, H3.2, H3.4 et H3.5)
    its PROVIDED INTERFACE (extracted from ODS)

its input and outputs (DATAFLOWS et EXCEPTION_FLOWS extracted from ODS) description of its dynamical behaviour (extracted from ODS OBCS fields).

- verification reports upon the design steps.

2.3.2.3 Summary on Documentation and Reviews

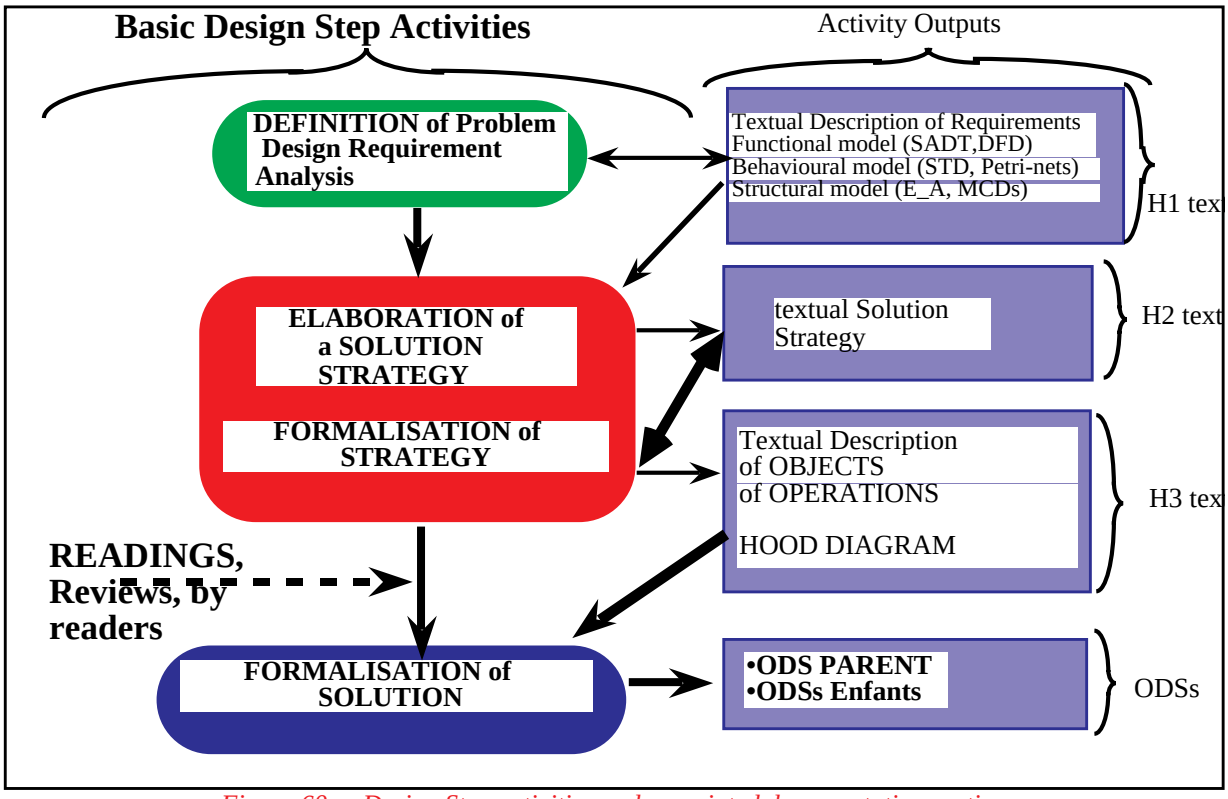


Figure 60 - Design Step activities and associated documentation section

Figure 60 - above summarizes the activities of a basic design step and illustrates the associated documentation sections to be produced and checked. We thing that the important things to recall and to enforce (by quality assurance) are:

- **good performance of the H1 activities,** which allows then to a designer to both i**ntegrate the requirements** and to outline associated solutions (or parts of) strategies in its mind.

  Note that the elaboration of H2 and H3 sections are not sequential activities, even if associated documentation sections are always presented as sequential ones. The description of a solution strategy can be produced in parallel and/or after an elaboration of a graphical outline, and vice-versa. The refinement of textual descriptions of operations and/or objects in section H3 allows to gain a better understanding of the solution, and hence and refinement of the earlier solution strategy produced as H2.

- **the central importance of an author-reader cycle at the end of H3 activities.** At that time, **a textual and graphical description of the solution is available in a form that is understandable by all project members.**

- Experience on earlier HOOD projects has shown that author-reader cycles on more formalized notations was almost useless, and that **short author-reader** cycles are mandatory, with

a low volume of information to be checked[10]. **The end of H3 is the ideal time[11] for a efficient reading of the solution.**(later it will often be too late to impose changes in the solution)

- Moreover, the next phase of formalisation of the remaining fields of ODS is a work that will require much more efforts, and which can only be checked by motivated people[12] which are specialized ones in the notations used (Ada, C, STDs, or C++)
- **the importance of the understandibility of the texts produced in sections H2 and H3.** Experience has shown a bad tendency for HOOD designers to produce cryptic H2 and/or H3 descriptions. (often because they do no really see why they have to produce these text sections, because they always think the important thing is the code.). We take her the opportunity to recall that:
  - "**good design is something that reads clearly**[13]"
  - **most of these texts may be extracted** to setup a design document **for a formal project REVIEW**, where people from different application areas may participate. In that case, these **descriptions in section Hi are primary elements for quality and complexity mastering assessment** of the system under design.

2.3.3 DESIGN STEP VALIDATION

Verification is primarily fulfilled by informal quality assurance techniques such as inspections, walk-throughs and reviews over HOOD documentation produced during a basic design step. Since HOOD allows a designer to produce informal textual descriptions in natural language together with semi formal graphical descriptions, these can be easily reviewed through walk-throughs author-reader cycles. Experience has shown that those informal parts of design descriptions can be successfully reviewed before the formalisation into HOOD ODS and Ada_PDL begins. ODS formalised descriptions are very difficult to review by human readers and must be verified by specialized people with the help of design checking tools. Hence the verification process onto HOOD designs must be carefully planned and must be performed all along the design process.

At the end of a basic design step, design evaluation and design metrics computation can be applied for evaluation of a design solution with respect to alternative ones (see [KAFURA] fan-in and fan-out concepts, applied to the object model). Consistency checks can be performed with the support of tools, ensuring verifiable transformations to the target language and allowing early execution and testing of the solution model corresponding to a given parent. Within an Ada environment early verifications of interfaces is possible, especially parent-child signatures checks can be performed, and the mappings of requirement behavioural models into object control structure descriptions can be traced.

[10] otherwise, it is sure that reader will not have enough time to integrate the information and hence will not do an efficient job.

[11] Author-reader cycles on H2 may help a designer, but are globally less efficient (because reader are involved in the elaboration process). However the H2 section may be checked by all design team leaders (of a distributed team project) for the first top-level decomposition.

[12] for instance, people which will have to elaborate tests plans...

[13] When a design is expressed textually in a clear, readable form by the designer, it often means a well mastered solution. cryptic, non understandable texts shows complicated, non mastered solution in the mind of the designer.

2.3.4 DESIGN PROTOTYPING

Design prototyping can be performed at the end of a basic design step by producing executable models of children, on behalf of their OBCS descriptions. By comparing the behaviour of the parent upper-level prototype with the resulting one of current design step, and when applying the same test cases, one can validate the current parent-child decomposition.

2.3.5 LEVEL VALIDATION

When all validation step of a given level have been performed, level validation may be performed in order to check the consistency over several basic design steps together. This allows early identification of possible common objects in different branches of the design process tree. The definition of a design prototype by implementing each current terminal objects allows on another hand, to set up a test and pre-integration environment for each object with respect to its brothers

2.3.6 DESIGN VERIFICATION IN THE DEVELOPMENT

The above validation procedures enable **efficient distribution of the design** and development since:

- The scope of a given design step is limited to its nearest levels of decomposition according to the parent-child relationships. Each design steps produces specifications of objects which define as much new problems, and can possibly be validated by prototyping against either a behavioural requirement model and/or together with their brother's ones against their parent behavioural model.
- **Level validation prototype can be reused as test and pre-integration environment** for all the objects of a same level, i.e for all brothers of a same parent.

Production of validated specifications of both the objects and their test environment (i.e.the brothers prototypes) allows to subcontract their design and development to other teams.

2.4 REAL TIME

2.4.1 DEVELOPMENT APPROACHES

The classical development approach for Real-Time Systems (RTS) considers the **process as the unit of modularity**. HOOD on the contrary considers the object (a grouping of services executed by processes) as the unit of modularity! What are then the possible HOOD approaches when faced with a development involving real Time constraints?

- **CURRENT APPROACH**
- The current approach models processes as HOOD objects and expresses classically a design as a set of HOOD objects/processes which interact through OS communication and synchronisation mechanisms. Figure 61 - below gives an example of a HOOD object modelling a real time process.

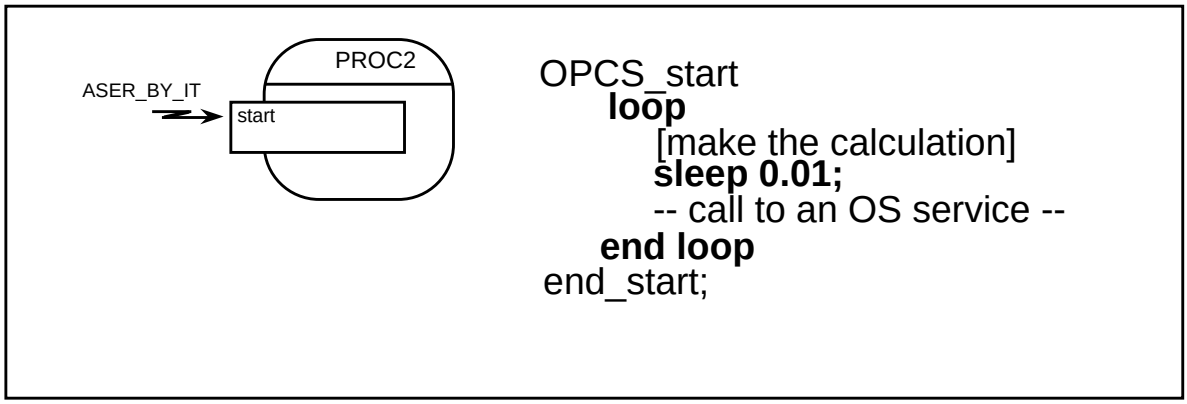


Figure 61 - Representations of tasks with HOOD

- **ADVANCED APPROACH**
- This approach tries to reconcile the structuring approach in terms of objects with the structuring approach in terms of processes, by considering a system as a structured set of *heavyweight* and *lightweight* processes, as suggested in [Ada9X and Mull90].
  - *Lightweight processes* (threads in some targets see last revision of UNIX V) are implementations of logical processes that execute HOOD operations. Thus these processes are used to implement light synchronization to access data, mutual exclusion or monitors. Lightweight processes execute all in a same memory partition.
  - *Heavyweight processes* are OS processes, that define a virtual machine, and can be seen as defining a memory partition.with hardware protection on its boundaries (memory trap, violation, or remote machines). Interrupt driven tasks are also heavyweight processes because their scheduling is constrained by real-time constraints.
- The granularity of heavyweight processes will be the concern of real-time architects and can be supported efficiently by the concept of HOOD VNs. Hence the HOOD development approach for RTS should follow closely the phased approach recommended for complex systems with:
  - *logical definition and refinement phase* elaborating a logical solution (non partitioned) as described above.
  - *allocation phase* where Virtual Nodes are defined as grouping of objects by allocation defining potential memory partitions[7]. The allocation process starts by defining a possible

target architecture of the RTS as a network of VNs.:

- either by modelling the physical architecture when processes/threads are numerous and must be abstracted into less heavyweight ones according to the potential power of the target (e.g. max =127 process schedulable, but only 40 allowed in order to reduce scheduling load)
- either as a model of a planned architecture, if not yet defined (or available since feasibility studies are still in progress) at that stage of the project,
- or as a representation of the (heavyweight) processes, or partitions [7] if the hardware architecture was already frozen.This representation takes into, account efficiency requirements such as high acquisition rate processes, computing time, asynchronous behaviours (MMI, DBMS, CENTRAL PROCESSS), high priority tasks, interfaces to existing systems, interrupt-driven processes, etc.

- *configuration phase* where grouping and restructuring of VNs is done for allocating them the chosen configuration of heavyweight processes

2.4.2 CURRENT DEVELOPMENT APPROACHES

2.4.2.1 Representing Common Real Time Mechanisms

Representing real time mechanisms was often a problem, since these mechanisms are not explicit in Ada, and are generally used as services provides by real time libraries. In order to have an homogeneous representation of these mechanisms, whatever the target features, we recommend to model them through HADT objects. These latter define a logical interface invariant from one target to another, and from one project to another. Moreover these interfaces may directly map those provided by standard executive service packages such as EXTRA or CIFO proposals.

2.4.2.1.a Tasks and Cyclic task

A task is an implementation of an *abstract process type* providing start, stop, create and delete operations. Cyclic task may be represented through:

- either by a “ASER_BY_IT cycle duration” triggering the main activity operation
- either through a looping code that suspends itself on a timer

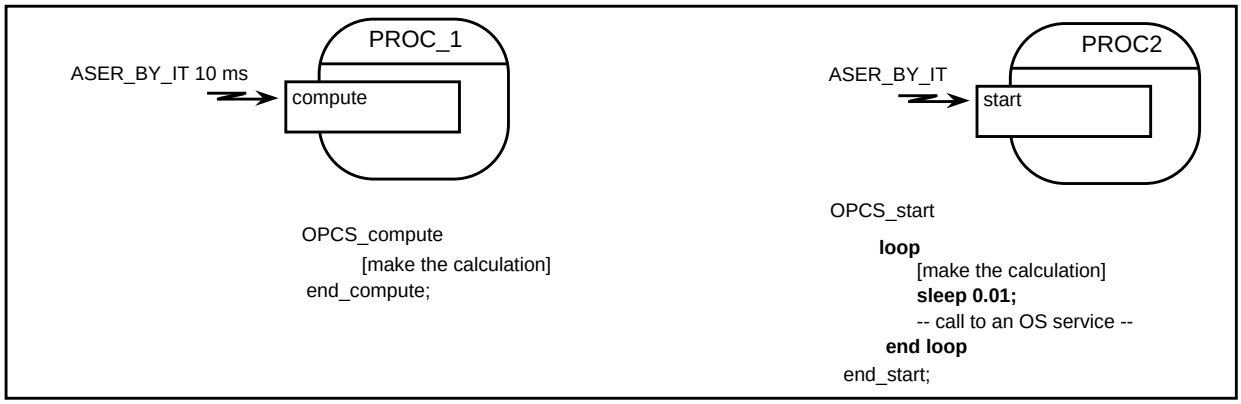


Figure 62 - Representations of cyclic tasks;

2.4.2.1.b Semaphores

A semaphore will be represented as an environment HADT object providing the operations to create, delete a semaphore, and to take a resource (P) or release a resource (V)

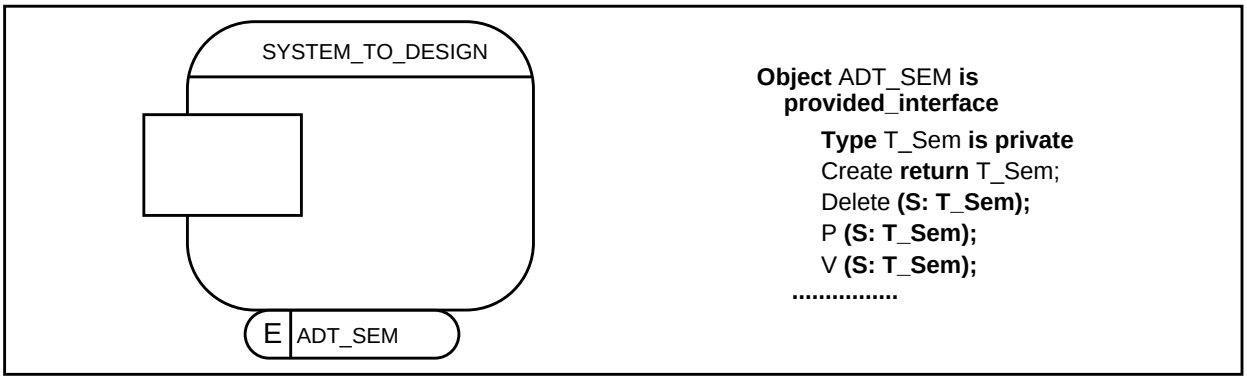


Figure 63 - Representation of semaphores

2.4.2.1.c MAIL BOXES

A mail box will be represented as an environment HADT object providing the operations to create, delete a mail box, and to take a message (GET) or put one (PUT).

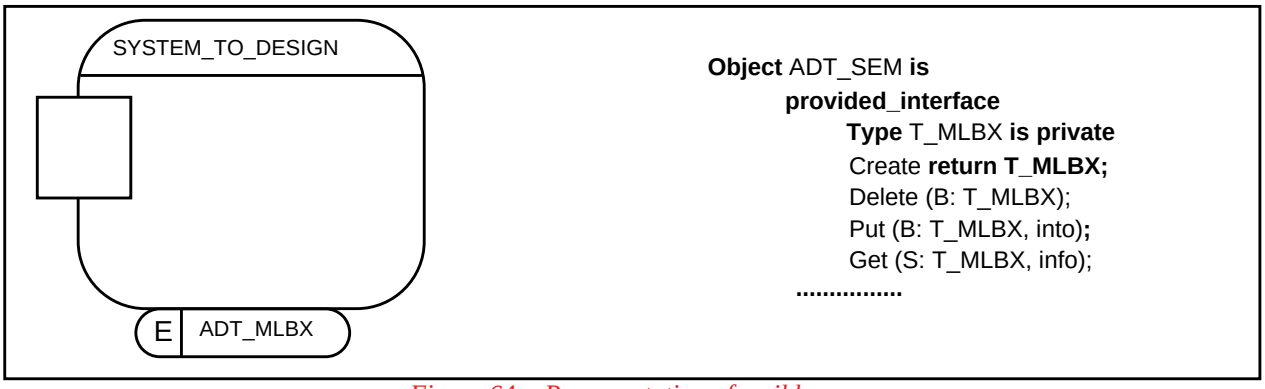


Figure 64 - Representation of mail boxes

2.4.2.1.d SHARED DATA

A shared data will be represented as an environment HADT object providing the operations to create, delete a shared area, and to write (WRITE) or read information from that area (READ).

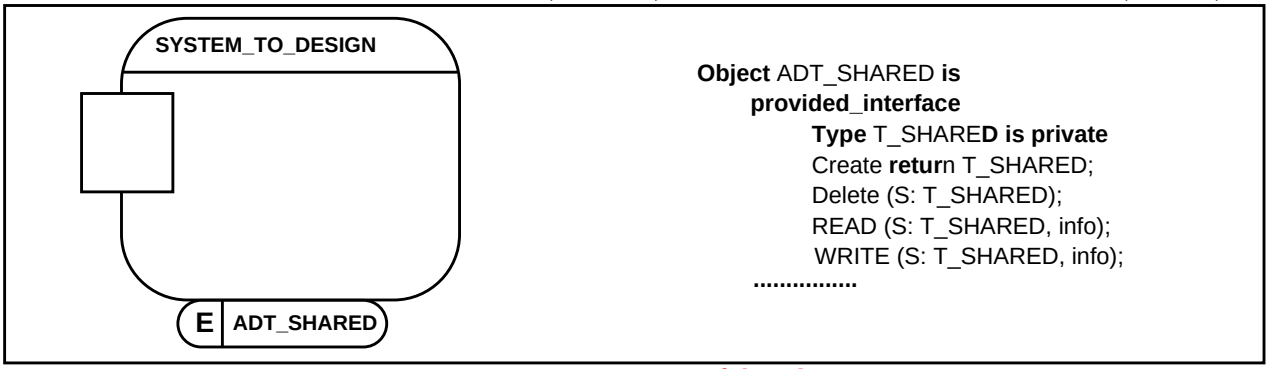


Figure 65 - Representation of shared areas

2.4.2.1.e EVENT

An event will be represented as an environment HADT object providing the operations to create, delete an event, and to set (SET) or reset (RESET) or wait for (WAITFOR) an event.

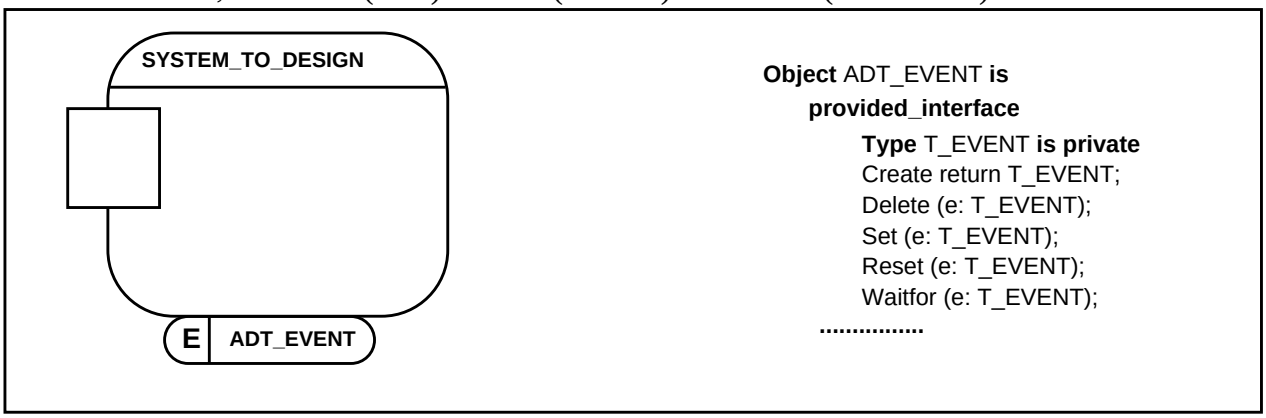


Figure 66 - Representation of events

2.4.2.2 Establishing a Real Time architecture

Let us take an example of a system of a three process system (targeted say to the VMS OS). A HOOD architecture can be established according to:

- the allocation of functionalities and activities logically related to a process.
- process definition depending on efficiency considerations (e.g fast acquisition process, batch data processing, asynchronous MMI process, asynchronous Database servers, etc.).

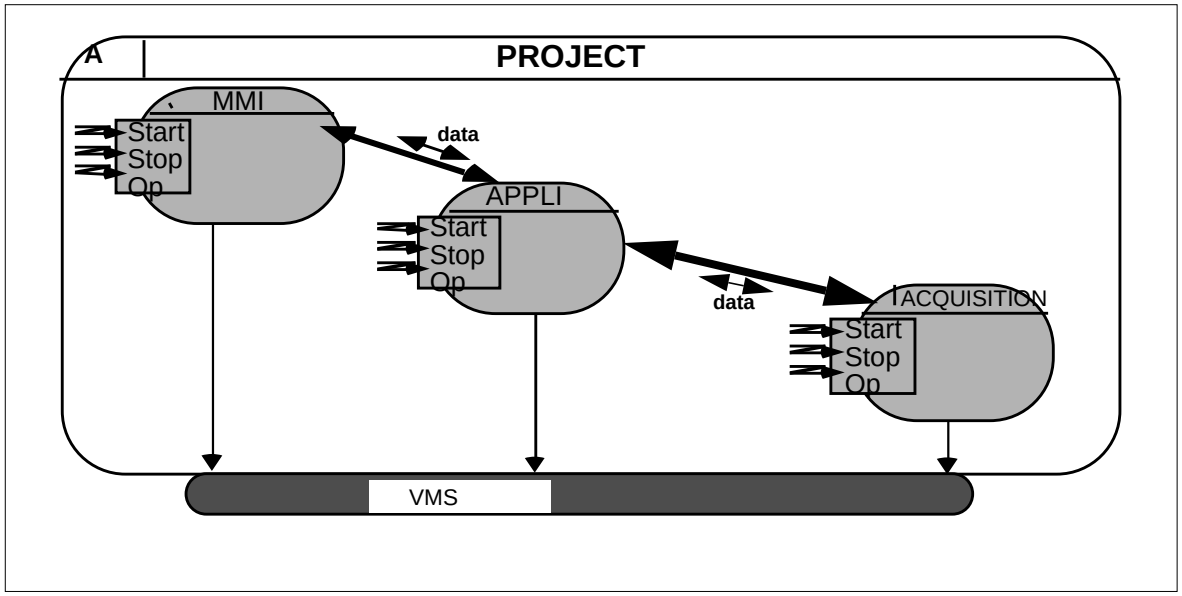
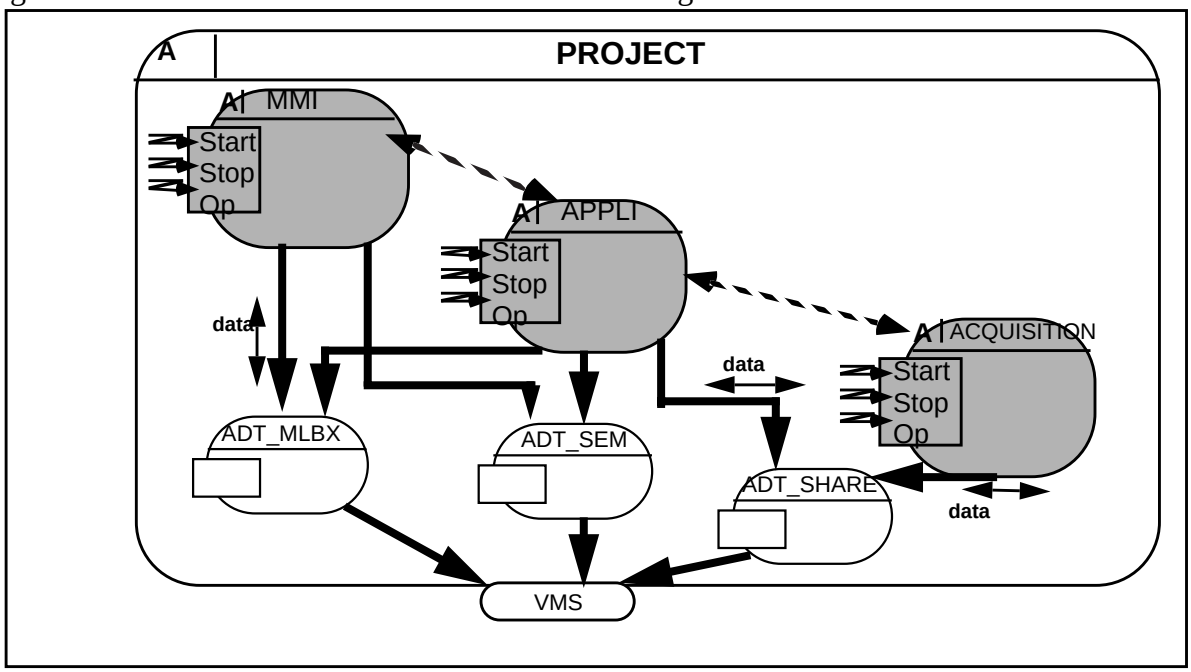


Figure 67 - Initial representation of a RT architecture

Figure 67 - gives an initial representation of a the HOOD architecture as a set three abstract process implementations. One sees at once that data flowing between two objects, will effectively flow through the inter-process communication mechanism, which is in our case implemented using the primitives IPC services of the VMS OS. In order to refine the definition of these communication the next HOOD refinement should explicit the use of theses mechanisms, in order to

better control these resources and adapt code generation.(e.g. if a mailbox MB was defined, two objects may communicate data through HSER by MB protocol constrained operations).
Figure below illustrates the HOOD refinement making VMS IPC communication services and



tools explicit.

Figure 68 - Explicating inter-process communications

From now on the initial objects can be refined and developed as in a classical real time development. All interfaces are explicitly specified by means of ADT objects.

2.4.3 ADVANCED DEVELOPMENT APPROACHES

The main question here is "*should a designer consider the software of a system to design a set of task or as a set of OS processes (programs) first*?" The answer is "*it depends on the importance of the non functional constraints*" of the given projects:

- If heavyweight process architecture can be easily established (only a few OS process, established already from requirements analysis) then an initial architecture can be directly established first.
- If however the requirements analysis identifies numerous tasks, whose grouping is not obvious, the VN architecture should only be established after a definition of a logical solution defined as a set of HOOD objects.

2.4.3.1 Establishing a VN architecture

Let us take our example of a system of a three VMS process system. A HOOD VN architecture can be established according to:

- the allocation of functionalities and activities logically related to a VN. (If a logical solution has already be produced, activities are allocated to objects and can be mapped into a VN by allocating the objects).
- the allocation of functionalities which are quite independent of others already mapped into

others VNs; this allows to define VNs as work-package for parallel developments.

- VNs definition depending on efficiency considerations (e.g fast acquisition process, batch data processing, asynchronous MMI process, asynchronous Database servers, etc.).

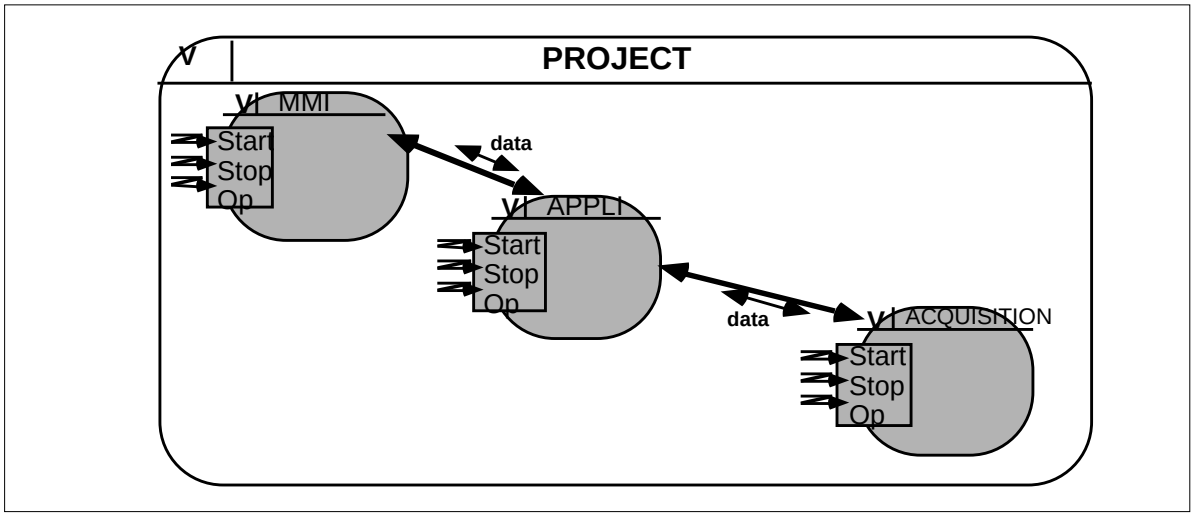


Figure 69 - initial representation of a VN architecture

Figure 69 - gives an initial representation of a VN Architecture. One sees at once that data flowing between two VNs, will effectively flow through the inter-VN communication mechanism, which is in our case VMS. There are two possibilities for refining the definition of these communication:

- leave the code generator for VN do its work and generate appropriate code making use of the VMS inter-process mechanism to implement protocol constraints attached to the VN/OS processes
- explicit the use of theses mechanisms, in order to better control these resources and adapt code generation of allocated HOOD objects onto VNs to effectively use these mechanisms (e.g. if a mailbox MB was defined, two objects may communicate data through HSER by MB protocol constrained operations).

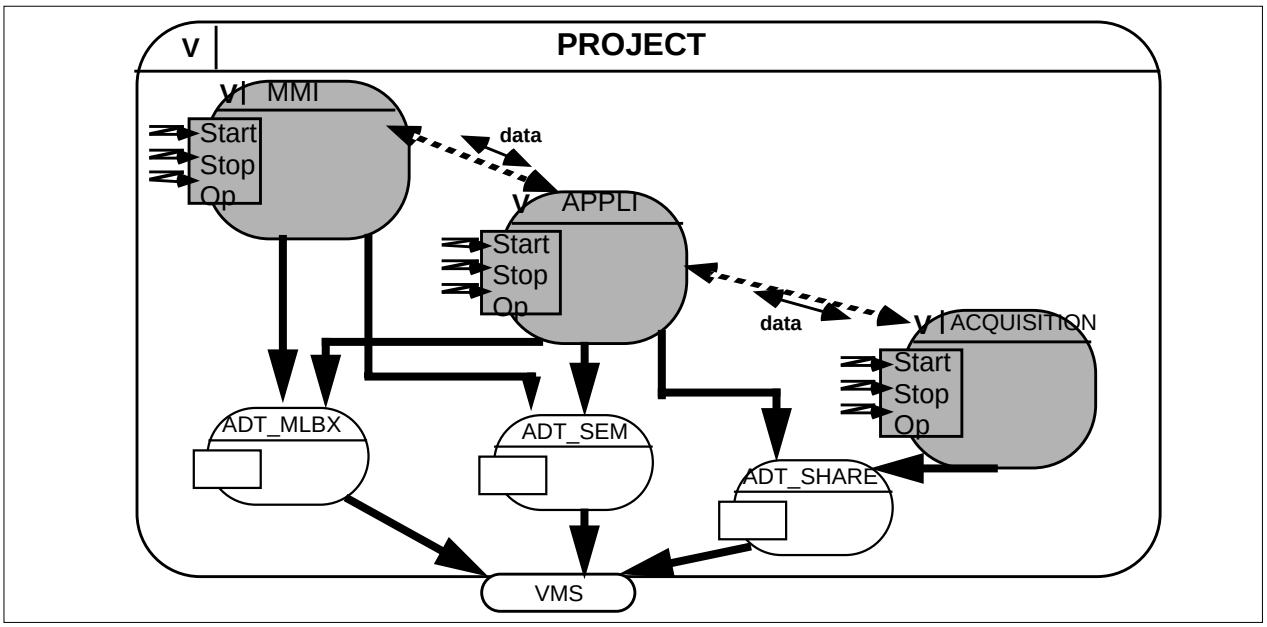


Figure 70 - Explicit inter-VN communications

2.4.3.2 Implementing INTER-VNs communications

The implementation principle inter-VN/OS processes communication is based on leaving state constraints implementation always within the server process executing a constrained operation. Hence the code structure for a VN becomes:

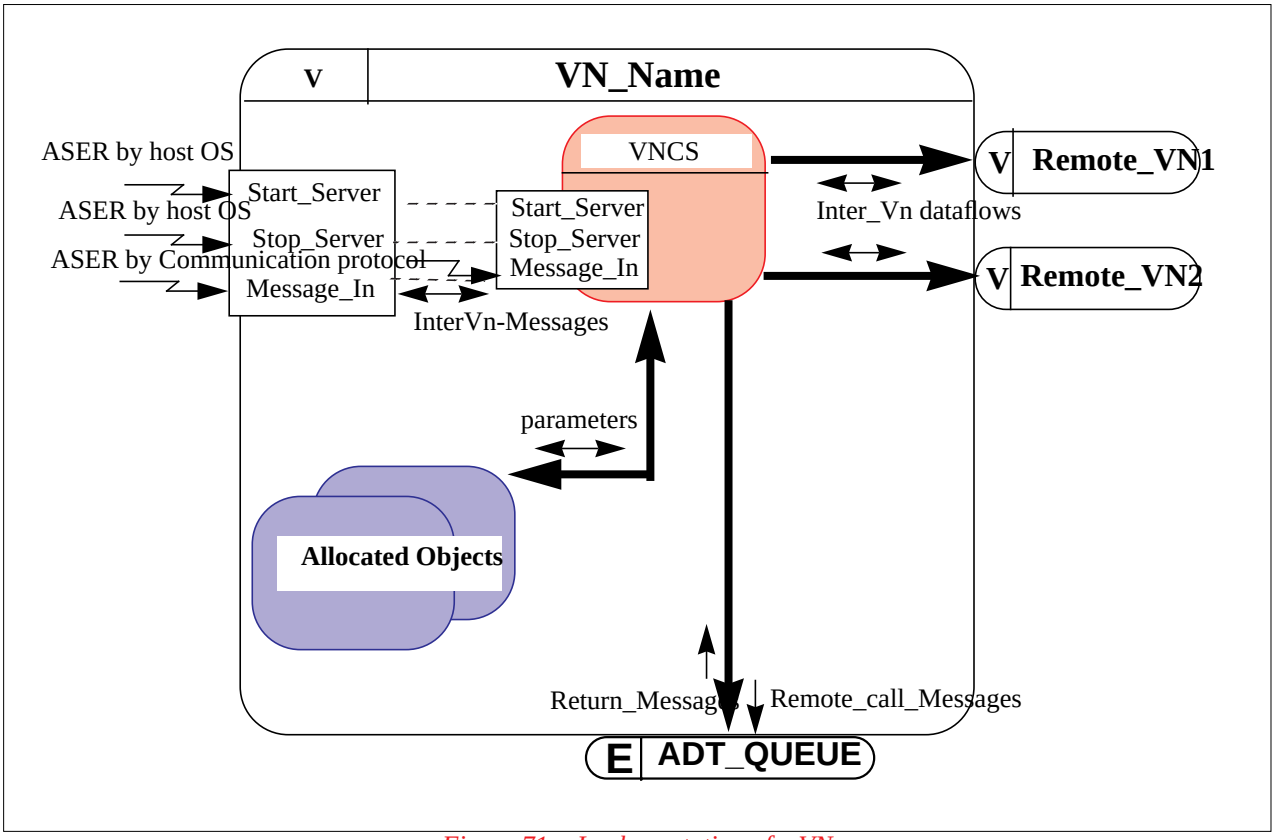


Figure 71 - Implementation of a VN

The figure above shows that all allocated objects, that required remote operations should in fact require a VNCS.Message_in operation that communicates with remote VNS by means of ADT_QUEUE

In case code generation is not fully automated in a given toolset or even not implemented, the designer would have to explicit every VN in his VN architecture. This solution gives full control (including over the generated code bases on standard HOOD objects) over the design of the communication mechanism between VNs.

2.4.4 EXPRESSING INTER-PROCESS COMMUNICATION WITH OPERATION CONSTRAINTS

HOOD allows to expresses communication protocols upon service execution of objects by assigning protocol execution constraints to operations. HOOD has only defined the following logical communication protocols:

- ASER: the client process is not suspended at all after a request
- HSER: the client process is suspended up to the end of requested service execution
- LSER: the client process is suspended up to the begin of requested service execution
- TOER: the client **wants to know** if a request was executed within a time delay.
- ASER_BY_IT: the client is the hardware, and this is the way to model interrupts.
- PROTECTED[14]: the client executes in mutual exclusion

A designer must notice that these are basically logical communication schemes between two processes, and that associated implementations (code generated) may furthermore use the most common communication mechanisms available from the target host system:

- rendez-vous
- RPC
- semaphores
- mailboxes and queues
- signals
- shared areas.

How can a designer used to express inter-process communications with these mechanisms express them in HOOD? In fact there are two possibilities:

- expressing a HOOD design that directly reflects the implementation
- In this case, the used mechanisms should be represented as HADT objects
- expressing logical communication at HOOD level, knowing that:
  - code generation of HSER,LSER may achieve the same effect as for example two object communicating through a third one modelling a QUEUE.
  - allocating objects on VN that define inter-VN communications, whose communication mechanisms will be expressed with target communication mechanisms.

Depending on the target system chosen (either Ada tasking supported or not) and according to the code generation principles, one can state the following rules.

[14]. such constraints are not in current HOOD definition, but are a proposal that came out from the Hard Real Time study funded by ESA.

2.4.4.1 Use of Ada tasking

- state constraint applied to an operation =>one task to ensure state constraints
- **protocol constraint applied on a operation =>** 2 process at least implied.
  - a client task performs a calls to the OBCS entry via the call to the OBJECT.Operation.
  - an OBCS task server (or more if needed) that:
    - accepts operation entry requests according to the protocol constraint[15] and
    - performs effective call to the an OPCS procedure named OPCS_<OP_Name> (that includes opcs-header handling possibly state constraints, and opcs_body with the core operation). and
    - return parameters within the accept statement

2.4.4.2 No use of Ada tasking

- **state constraint applied to an operation =>**
  - one FSM to ensure check STD description, but
  - possibly call to an OS semaphore to ensure protected access
  - need to have an implementation of exceptions in language other than Ada
- **protocol constraint applied on a operation =>** 2 process at least implied.
  - one FSM to ensure check STD description, but
    - a client task performs **an OP_ER procedure** part commuting the message request and decommuting the return parameters, and exception handling
    - a server (or more if needed) that:
      - consumes messages requests and
      - performs effective call to the OPCS procedure (that includes opcs-header handling possibly state constraints, and opcs_body with the core operation). and
- commutes return parameters in a MSG back to client task

[15]. Note that ASER constraints are badly supported in Ada83, and need additional server and consumer task to perform according to the HOOD semantic.

2.5 DISTRIBUTED SYSTEMS

2.5.1 A DEVELOPMENT APPROACH

Putting the principles of the overall HOOD design process for complex systems into work for the development of real time or distributed applications leads to a phased development approach comprising three phases:

- *Logical definition and refinement phase* elaborating a logical solution (non distributed) as described above.
- *allocation phase* where Virtual Nodes are defined as grouping of objects by allocation defining potential distributable units or memory partitions[7]. The allocation process starts by defining a possible physical model:
  - either by modelling the physical architecture when processors are numerous and must be abstracted into processing nodes according to the characteristics of their communication links,
  - either as a model of a planned physical architecture, if not yet defined (or available) at that stage of the project,
  - or as a representation of the physical (heavyweight) processes, or partitions [7] if the hardware architecture was already frozen.
- *configuration phase* where grouping and restructuring of VNs is done for allocating them onto physical processors. The configuration must deal with network constraints and target inter-processors links and communication features.

2.5.2 VN IMPLEMENTATION APPROACH

The implementation approach is based on principles enforcing the reuse of the code developed in the logical model “as-is” (also called Post-Partitioning Approach in the literature). The latter supports a stepwise development and integration approach, allowing functional testing, and final integration leaving functionally tested code unchanged. Hence the execution of a remote operation can be implemented by defining:

- on the local node, a client operation stub that encodes the parameters into a network message, say MSG, and waits operation parameters return according to the communication protocols defined between the two remote objects.
- on the remote node, a server will decode the network messages and perform the effective call to the operation, and return parameters back through a special return_message network. This return message is then processed by a message_server from the original calling node and parameters are send back to the original client object[1].

Figure 72 - below illustrates the principle of adding code to existing code (generated in a non distributed or logical solution and functionally tested).
When an object is allocated to a VN, the generated code associated to the local OPCS consists e OPCS_ER (**O**pcs_**E**xecution **R**equest) code and performs the encoding of the request into the

[1]. This is done by having the associated process hold on a signal, which is sent as soon as the return parameters have been copied back to the client process memory.

network message suitable for the communication system (either simple UNIX sockets or higher level tools).
On the remote VN side, a local “server”, (*the ServerObcs*) tailored to each VN, recognizes the messages that come from the network and performs the effective call to the operation code. When the operation has executed, a “return-message” is sent back (through the *ServerObcs*) to the calling VN and object.
Three solutions may be used to achieve such an implementation, and leaving the original code unchanged:

- using a HOOD toolset that supports fully the implementation for VN.
- generation of additional code outside from HOOD, and depending of the allocation of objects onto VNs, and building of a suitable executable by doing the appropriate linking of the object code.
- generation of additional code from a HOOD additional representation, that shows the allocated objects in a «VN design». This solution will be further developed in section 2.6.1 below as today no HOOD toolset supports fully VN code generation.

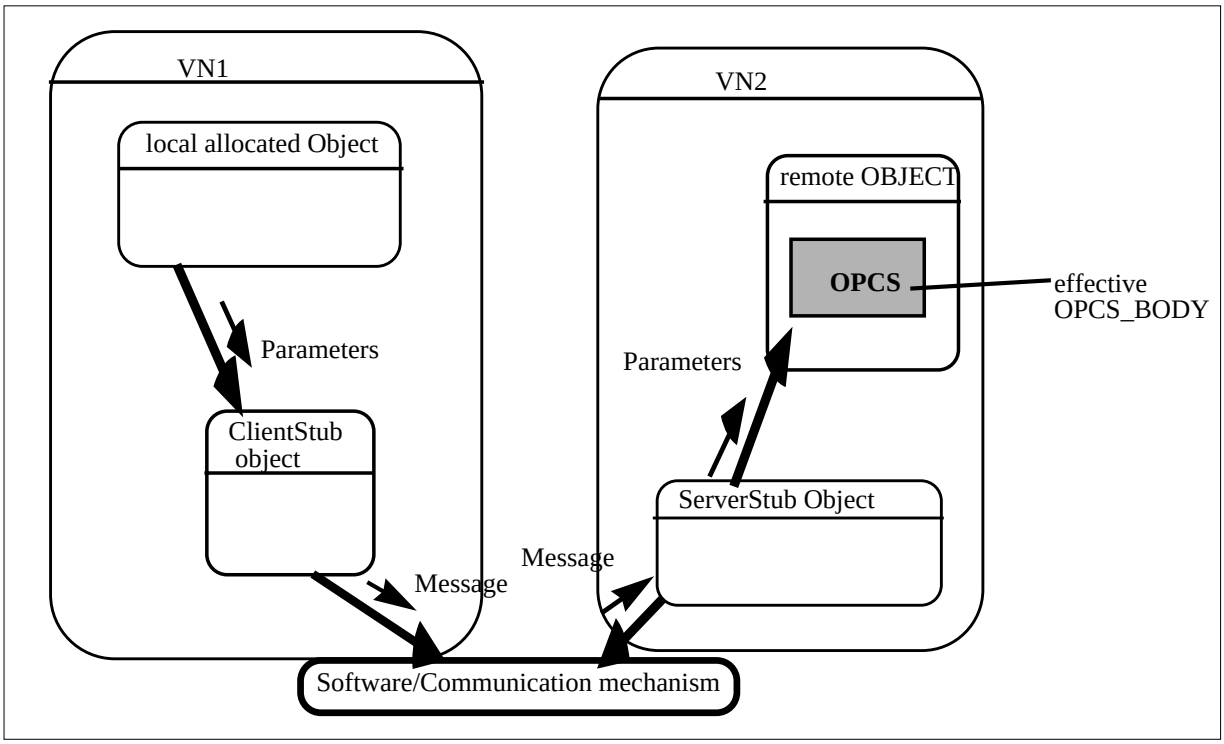


Figure 72 - Principle of additional code to the logical model one in the physical model/

2.5.2.1 Implementing protocol constraints for VNs

Figure 73 - gives the principle of code generation for VN, *reusing and leaving unchanged the functional code generated of the logical model* (with no protocol constrained operations).
The ClientStub object of Figure 72 - above is implemented as a set of objects (as many as allocated objects to the VN that require remote operations) and whose OPCS code contains only OPCS_ER code as illustrated in Figure 74 - below. This implementation is heavily based on *FIFO queues*, of which efficient implementations are now available for most targets. Queues allow to handle the problem of global time, network and process contention, synchronisation, etc..

in the most modular way. Furthermore it allows to have a layered implementation decoupling fully the HOOD application of any implementation infrastructure and communication software.

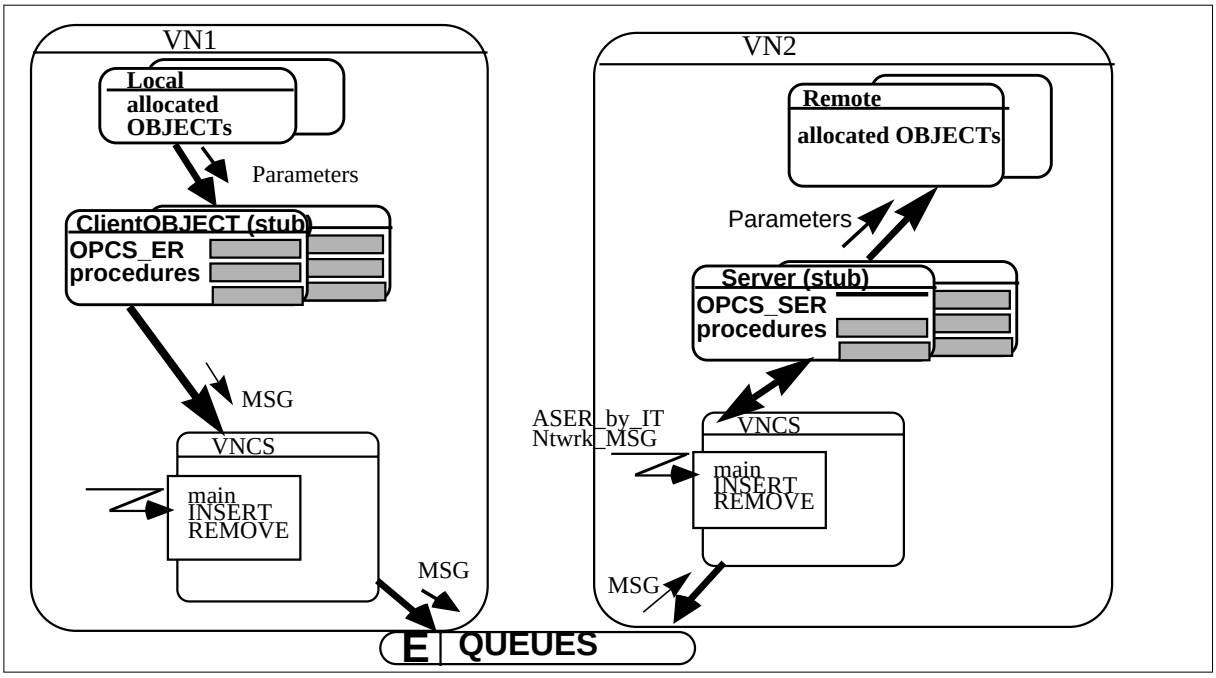


Figure 73 - Execution Model for protocol constrained operations for VN

```
procedure PUSH(item : in T_Item) is -- OPCS_ER code
MSG : IPCMSG.T_MSG;
begin
   MSG=IPCMSG.create;  - initialise an IPC MSG data structure
   MSG.sender=STACK;
   MSG.OPERATION=PUSH;
   MSG.CNSTRNT=HSER[1];
   MSG.INFO=Item;
   ClientObcs.insert(MSG); -- insert in request queue
   -- if firstimeCall queues are allocated
   MSG:=ClientObcs.remove; -- remove return IPCMSG and process return parameters
   if not (MSG.X=OK)then
      EXCEPTIONS.raise(MSG.X); --raise exception according to Xvalue
   else
      EXCEPTIONS.LOG ("STACK.PUSH_ER", "OK return from server")
   end if;
   if not (MSG.CSTRNT=ASER) then
      ClientObcs.FREE(MSG); -- deallocate MSG and queues after processing return parameters
   end if;
exception
when X_BAD_EXECUTION_REQUEST =>
   EXCEPTIONS.LOG("STACK.PUSH", "X_BAD_EXECUTION_REQUEST");
   EXCEPTIONS.raise;  -- so as to propagated to client
when Others =>
   EXCEPTIONS.LOG("STACK.PUSH", "Others");
   EXCEPTIONS.raise;
end PUSH;
```

[1]. if the request is also TOER, then we would insert here a call to TIMER.SET(delay), as well as a TIMER.TimeOut call after return (MSG:=ClientObcs.remove; -- remove return IPCMSG

Figure 74 - OPCS_ER code Sample for STACK.PUSH client stub operation

The ServerStub object of Figure 72 - above is implemented as a set of objects (as many as allocated objects to the remote VN that are required by other remote VNs) and whose OPCS code contains only OPCS_SER code as illustrated in Figure 75 - below.

```
procedure PUSH (MSG : in IPCMSG.T_MSG) is -- OPCS_SER code
Item : Items.T_Items;
begin
   begin
      item:= MSG.INFO;
      MSG.X=OK;
      if MSG.CSTRNT=LSER|LSER_TOER then
         ServerObcs.insert(MSG);  -- insert in return queue
      end if;
      STACK_SERVER.PUSH (Item);
   exception
   when X_BAD_EXECUTION_REQUEST =>
      EXCEPTIONS.LOG("STACK_SER.PUSH", "X_BAD_EXECUTION_REQUEST");
      MSG.X= string (X_BAD_EXECUTION_REQUEST);
   when Others =>
      EXCEPTIONS.LOG("STACK_SER.PUSH", "Others");
      MSG.X= string (OTHERS);
   end; -- of begin
   if MSG.CSTRNT=HSER|HSER_TOER then
      ServerObcs.insert(MSG); -- insert in return queue
   end if;
end PUSH;
```

Figure 75 - OPCS_SER codefor STACK.PUSH RB operation

The communication software between the VNs in Figure 72 - is encapsulated within a VNCS which is part of each VN. This allows to have OPCS-ER or OPCS_SER code that abstracts from the specificities of different target OS or communication software. Since a VN may be both a server and a Client the VN may be partitioned in at least two sub objects ClientObcs and ServerObcs as illustrated in Figure 76 -

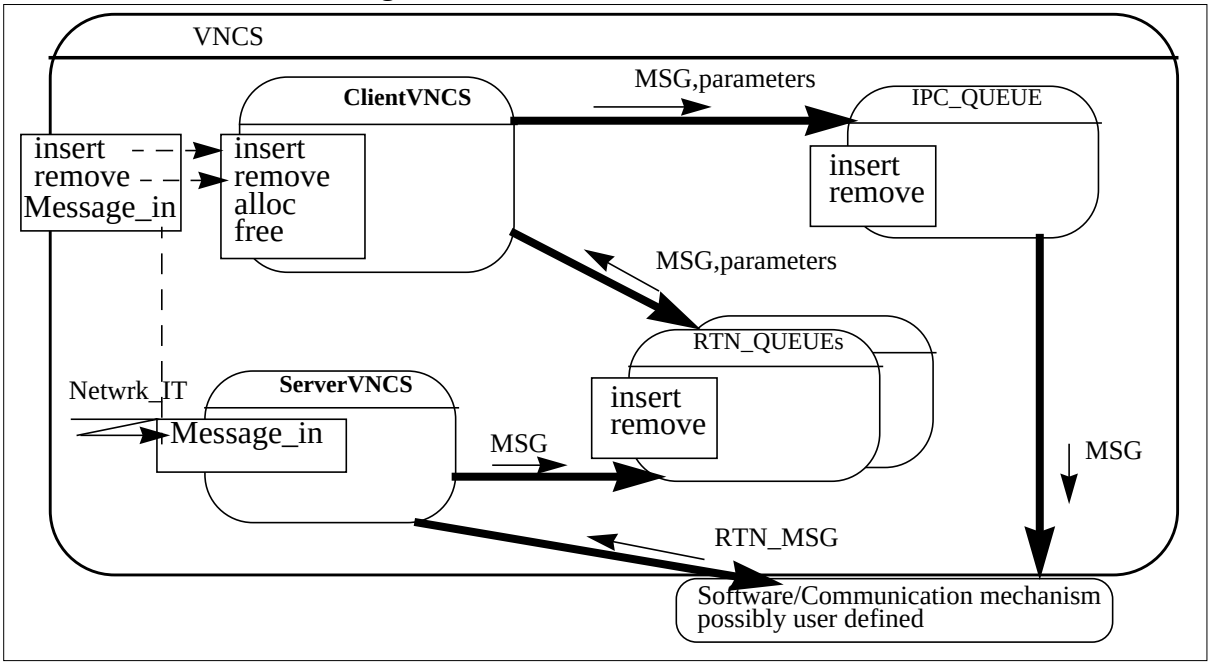


Figure 76 - Architecture Principle of the VNCS software

The operation ClientObcs.Insert builds up the IPC MSG Structure (especially does the *marshalling*[2] converting the parameters structure into streams, allocates a RTNQUEUE and puts its identification in the MSG, and is blocked on the return MSG by calling RTNQUEUE.remove operation. Messages incoming through the communication software (possibly user defined and provided) shall trigger[3] a Message_in operation of:

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

.

```
procedure Message_In(MSG : in T_MSG) is -- ServerVNCS code illustration
begin
loop
   case MSG.type is
   when RTN_MSG =>
        MSG.RTNQ.insert(MSG); -- insert in RTNQ will release client
   when IPC_MSG=>  -- just launch local OPCS_SER code
        MSG.IPCQ.insert(MSG); -- insert inIPCQ will launch ServerOBCS of called Object
   when others =>
                EXCEPTIONS.LOG (“VN_ID.Message_IN”, “INCONSISTENT MSG type”
                MSG.X=COMMUNICATION_ERROR;
                Message_out(MSG); --send messageback
   end case;
end loop
exception
when Others =>
   EXCEPTIONS.LOG(«VN_ID.Message_In», “Others”);
end Message_In;
```

Figure 77 - Illustration of ServerVNCS.Message_in code

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

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

```
    procedure ServerObcs is -- ServerObcs code illustration
    MSG: IPCMSG.T_MSG;
    begin
    loop
       ServerOBCS.remove(MSG); -- remove a message from a clientOBCS
       if MSG.CSTRNT=ASER then
            MSG.X=OK;  -- return OK EXCEPTION VALUE
            VNCS.Message_oput(MSG)); --send MSG back
       end if;
       case MSG.OPERATION is
       when start =>START; -- call procedure START of currentobject;
       when stop =>STOP;-- parameters accessed through global MSG structure
       when PUSH=>PUSH;
       when POP =>POP;
       when Status =>Status;
       when others =>
       EXCEPTIONS.LOG(MSGS.String(MSG;OBJECT)&»SERVER", "INCONSISTENT OPERATION NAME"
            MSG.X=COMMUNICATION_ERROR;
            ServerObcs.insert(MSG); --release client process immediately
       end case;
    end loop
end ServerObcs;
```

Figure 78 - Illustration of ServerObcs code of an allocated object in a remote VN

2.5.2.2 Managing VNS as HOOD OBJECTS

In the following we present an approach to define VN implementation according to the principles given above and allowing to use of the code generation defined for non Ada protocol constraint operations:
For users of toolsets not supporting VNs, the following steps should be performed:

- **Elaboration of a logical model with associated code generation and execution.** For this create a SYSTEM_CONFIGURATION with relevant hierarchies of objects, that defines the system_to_design with its environments objects. (objects common to those of STD hierarchies, as well as exiting and environmental software
- **Elaboration of the implementation model** (by allocating objects) **with the following constraints:**
  - only terminal VN can have allocated objects.
  - child objects are either the VNCS, the remote calling objects (generated with ERCODE), or the remotely called objects (generated with SERCODE).
  - Environment objects of this VN are either the local objects allocated or the remotely called VNS
    For this create a new SYSTEM_CONFIGURATION with its relevant hierarchies that define the system_to_design as a hierarchy of VNs. When the decomposition of VNs appears to be deep enough, allocate objects of the logical model to only terminal VNs[4].
    For each such terminal VN, create VNCS, ER and SER children and create environment objects that correspond to remotely VNS.
- **Finally create within the implementation SYSTEM_CONFIGURATION, root objects associated to the previous environment objects through "copy to root**" from objects of the logical SYSTEM_CONFIGURATION.
- Use then an ODS editor and modify the OPCS code fields so as to implement the encoding of operation request and associated parameters towards a network data structure and according to OP_ER template code ensuring the good behaviour with respect to protocol constraints.

Such a design is represented in Figure 79 - below. The conventions for this representation are:

- objects allocated to the VN appear as:
  - objects of same name but where all code generated shall be SERCODE
  - Environment object of same name which generated code is the effective code.
  - remotely called operations are highlighted by the implemented by links between VN operations and allocated objects. Such operations are eventually called by the VNCS object.
- objects remotely called/required from the VN appear as:
  - objects of same name but where all code generated shall be ERCODE. Such object eventually use the VNCS objects.
  - objects requiring remote VN that appear as Environment VNS

[4]. Note: a VN object can exclusively be decomposed either in child VNs, or in HOOD objects.

I

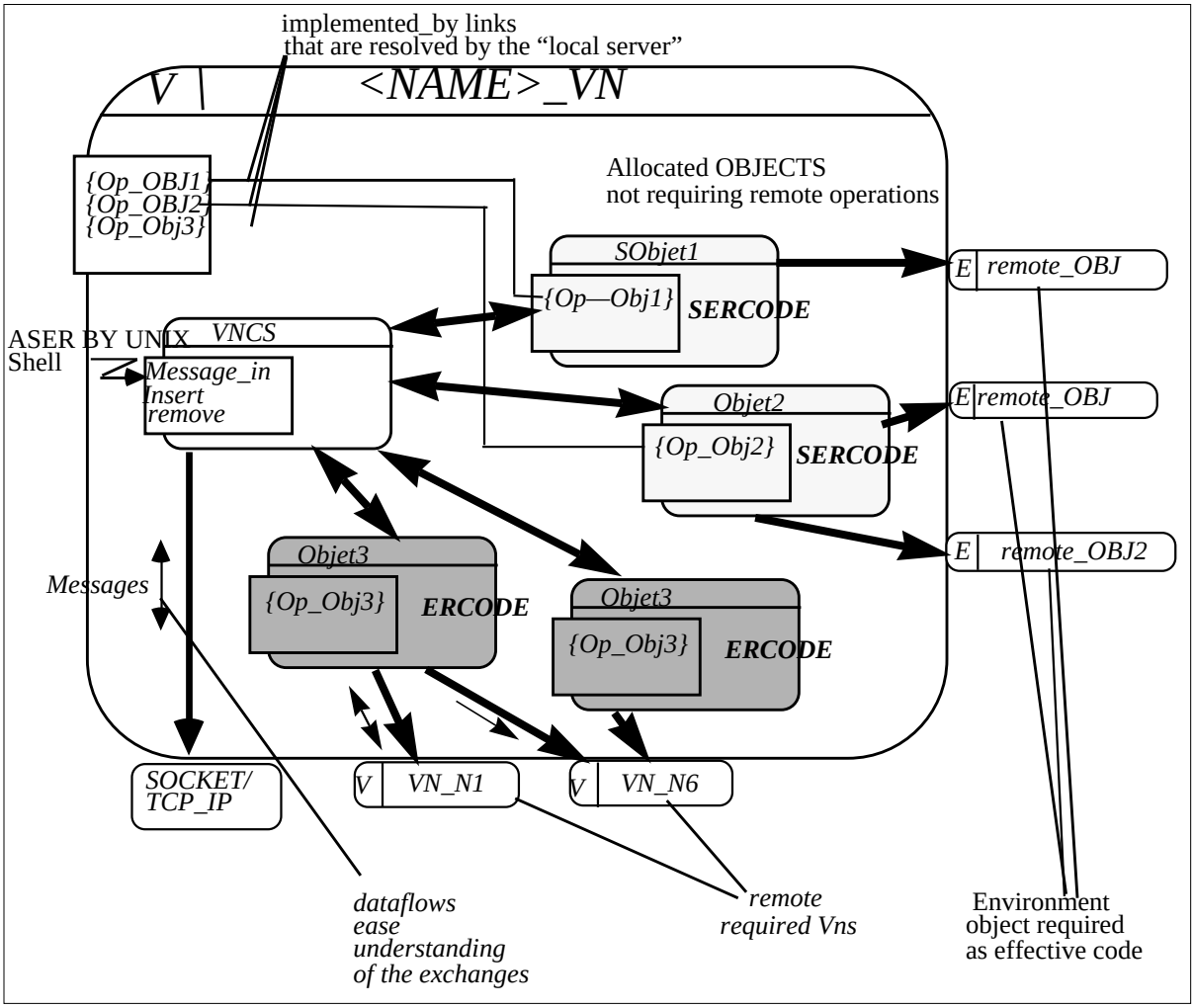


Figure 79 - Illustration of the representation of the physical model at terminal level

2.6 MAN MACHINE INTERFACES

2.6.1 DEVELOPMENT APPROACH FOR COMPLEX MMIS SYSTEMS

A common recommended approach for the development of Man Machine Interface (MMI) is to define an architecture allowing:

- separated development of the interactive software (which is handled by specialized windowing systems such as X_WINDOWS, MOTIF) and the applicative software, in a way that is independent of the technologies used for these developments. Moreover the interactive software may itself be spitted in a static part (now more an more automatically generated from MMI generator tools) and a dynamic part (mainly consisting in the code associated to the callbacks triggered by the window gadgets).
- the use of ad hoc technologies, people and tools with the best efficiency and productivity.

Although numerous developments have tried to follow this goals, effective separation of applicative and interactive parts has so far (to our knowledge) never or very few been performed; hence the resulting software are complex and costly to maintain.
The reason is that the interface between a software and a windowing system comprises both static, and dynamic entities and data, and, according to the two possible control strategies, (*event driven* recommended by the MOTIF standard and *application driven* recommended for reliable and critical systems) the associated code is distributed within the different objects/modules of the applicative or interactive part and leading to high efforts in testing and maintenance.

HOOD allows to represent easily interactive programs as separated objects exchanging data (e.g. APPLICATION and MMI objects) in an initial model. Those data may then be specified and refined as abstract data instances described in other HOOD objects hierarchies (following principles described in section 2.1 above).The clear separation formulated in this initial model allows to define three development lines:

- **development of the application part by** HOOD decomposition of object APPLICATION. The associated refinement may also provide for identification of operations provided of abstract data type objects associated to the data exchanged with the MMI part.The application object should provide the operations defining an API (Application Programming Interface) thus allowing to define application that can be interfaced either to other applications (batch, network oriented IO) or any windowing system.
- **development of man-machine part by** using suitable graphical interface generators and prototyping tools, and by “re-engineering” some of the associated software (namely all user-defined procedures associated to “call-backs”) into HOOD objects. A «call-back» is simply represented in HOOD as an «ASER_by_IT» constrained operation.In order to manage the code associated to window widgets call-backs, the designer can structure it into objects mapping either all the call-back of one widget hierarchy (e.g a window and all menus and attached buttons) or one specific wet of widgets (one HOOD object for an whole editor interface widgets).
- **development of data representation support** by defining abstract data type HOOD objects and target language type structures as the data and operations on those data are identified through refinement of APPLICATION and MMI objects. The associated formal definition of data allows to synchronize the development progress with the two other lines.

A recommended HOOD development approach suggest the definition of a generic architecture where these interfaces are clearly identified as illustrated in Figure 80 - hereafter. This architecture is moreover in line with the so-called Seeheim model [PFA85] where :

- presentation software is implemented_by the Windowing-INTERFACE and MMI_DATAs objects
- Application interface is implemented_by ADT_DATA
- DIALOG control is implemented_by DIALOG AUTOMATA objects

The Windowing_INTERFACE object handle all interactions between APPLICATION and the target WINDOWING system including behavioural execution according to finite state automata modelling the dialogue behaviour and states (that was defined in a requirement analysis or MMI prototyping phase).The MMI part can then be entirely prototyped using an UIMS: User Interface Management System.

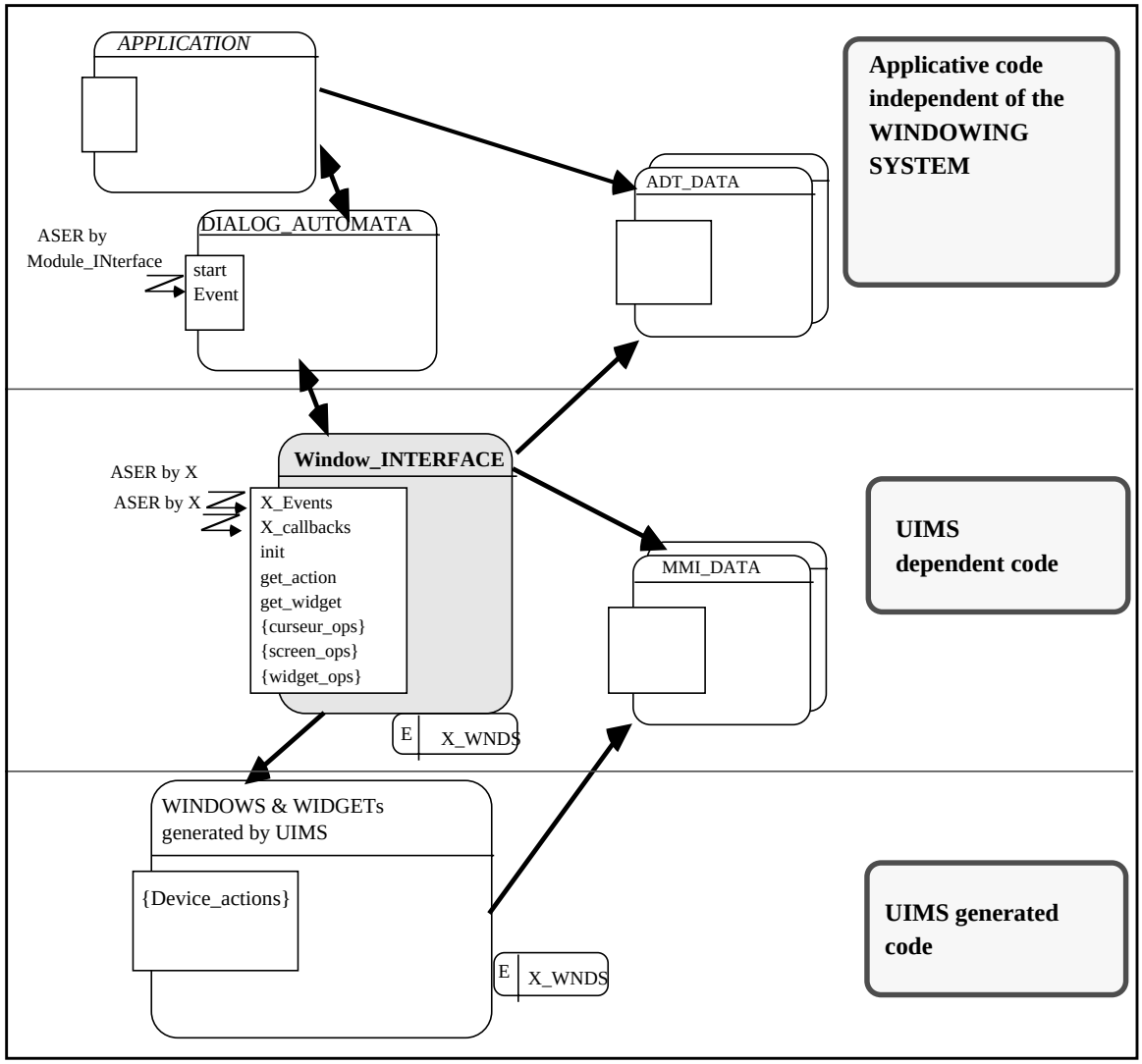


Figure 80 - Generic Architecture for MMIs

The DIALOG-AUTOMATA object should map the user-intercations modelled through a hierarchic state-transition diagram : the MMI interactions can be structured through states (e.g. logging,identification, consultation, update, supervision, administration, etc..) where only specific functions and widgets can be activated.

Such modelling and definition may be developed through prototypes or following a hierarchical Dialog structure approach, and possibly together with the final user. However we consider this modelling is part of the requirement analysis and should be available when the HOOD design is started.
Also, the above generic architecture allows to develop a MMI prototype (not to be confused with above one used for defining the user MMI interactions) where the animation may be thrown away or reused for pre-integration testing, but where DIALOG_AUTOMATA and Windowing_INTERFACE will be fully reused in the final software

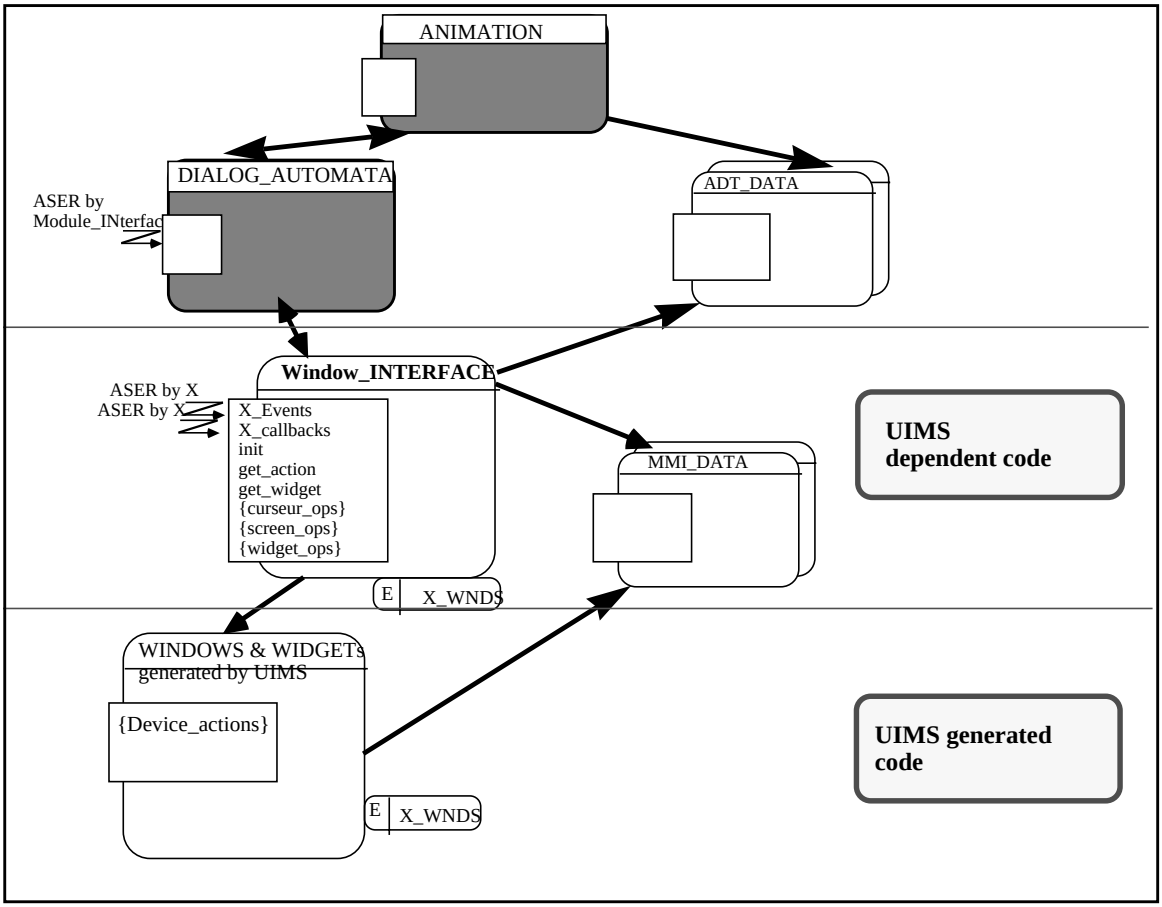


Figure 81 - Animation and prototyping of MMIs

The MMI development process should then comprise the following steps:

- **analysis** of OPERATOR behaviours and tasks, where requirements should be detailed enough to become the user guide of the future system. Besides a prototype of the MMI may be develop in order to get early feedback and acceptance by the user.
- establishment of associated **dialogue and screen definitions** by means of hierarchies of finite state automata. A HOOD toolset can be used be used to capture this states as HOOD objects where (see Figure 82 - below):
  - *start* would be the object to enter the initial state
  - *exit* would make the object go back to the calling state& automata
  - *help* would make launch a contextual help widget/object
  - *actions* are applicative actions to be triggered by the Windowing-INTERFACE as the user activates the widgets representing the actions allowed for the current object/state.

- **design of screens and widgets** under the UIMS
- **in parallel HOOD design** of the interface between the application using HADTS objects to formalise the data structures and the interactions with the non MMI part.
- Prototyping and unit testing
- evaluation of the MMI, modifications
- integration testing.

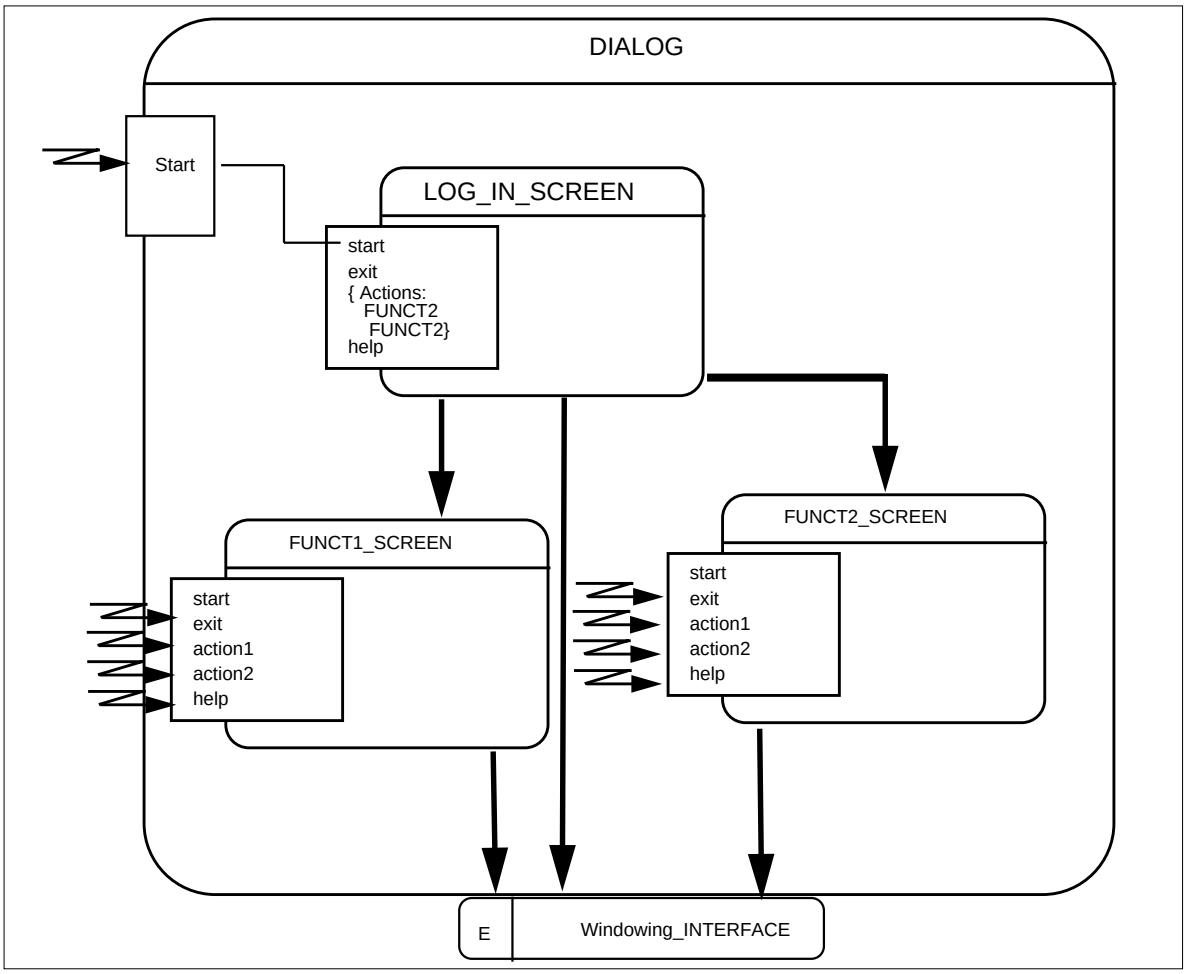


Figure 82 - Modelling DIALOG AUTOMATA with HOOD object

2.6.2 MODELLING INTERACTIONS WITH WINDOW MANAGERS

Graphical and interactive handling principles have been defined and summarized for implementation with windowing management systems (Sunview, Dialogue Manager, X-Windows and MOTIF). These latter provide Graphical User Interface Management Services (GUIMS) (by means of libraries of routines) allowing a client software to:

- manage graphical objects: creation, external representation, attributes, behaviours, reactions to operator stimuli of “WIDGETS” (contraction of “window gadget”, a graphical object whose is fully handled by the GUIMS)
- triggering of “application software” on WIDGET events (associated to an [re]action at widget). (these triggering are called “call-back” in the X_WINDOW system)

Thus the interaction logic between an application part and the window manager can be summarized as:

- initialisation of graphical interface through calls to the window manager services
- management of events (mostly associated to user interactions)by the window-manager
- triggering of execution of applicative part through “call-back” which are represented as HOOD “ASER_BY_IT” constrained operations.

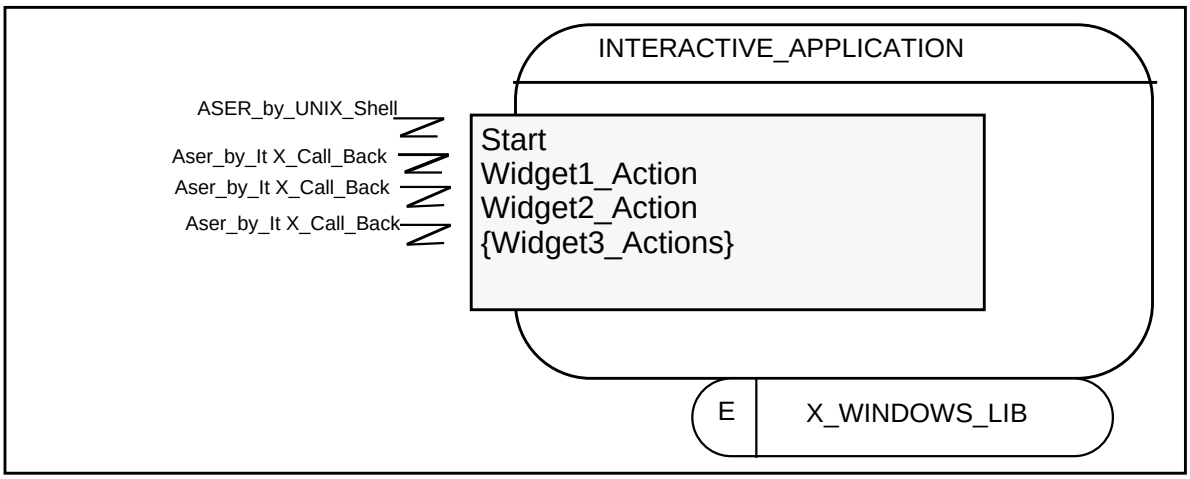


Figure 83 - HOOD Representation of callbacks

Figure 83 - above illustrates interactions between an applicative object and the X_WINDOWS window manager represented as a HOOD environment object.

2.6.3 FACTORISING INTERACTIONS WITH WINDOW MANAGERS

In order to structure and isolate the X_WINDOWS interface, thus rendering the architecture more portable and independent from the window-manager, a general interface can be defined. Such an architecture brings more control and flexibility on associated developments since the application can be developed, tested and pre-integrated in parallel to the development of the GUI.

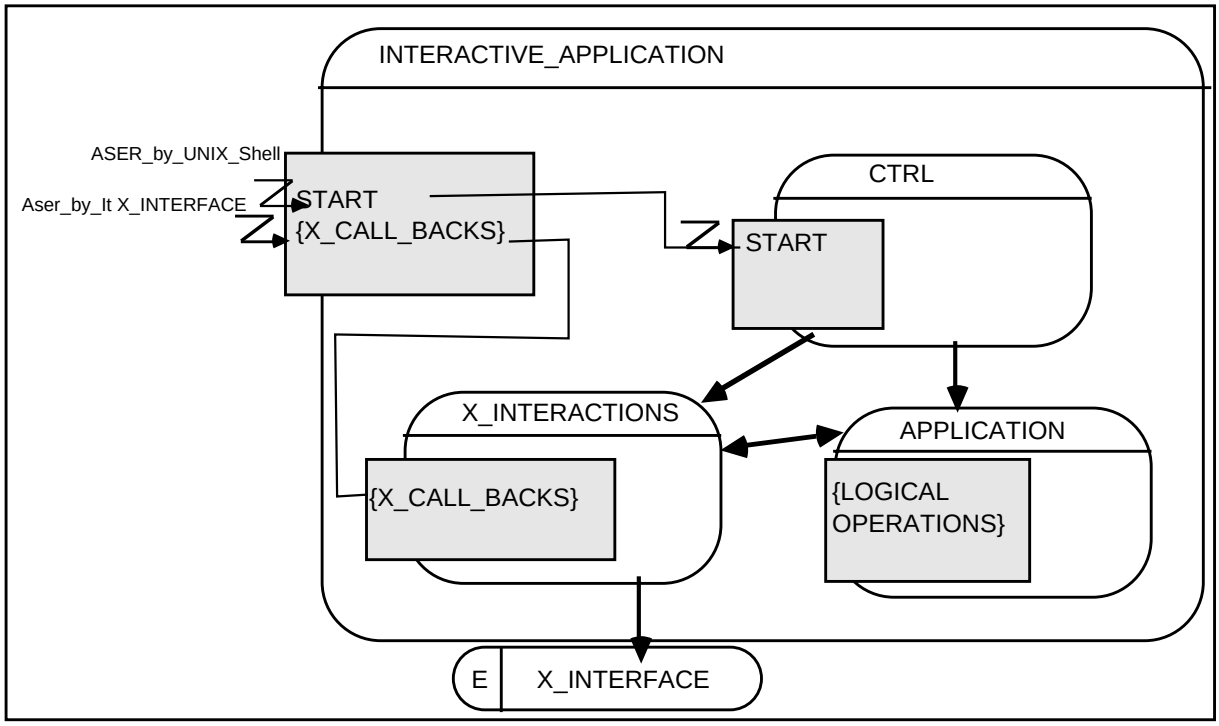


Figure 84 - Isolating the application from GUIMS code

The object X_INTERACTIONS handles fully interactions with the GUIMS, and may almost be fully developed within a GUI generator environment. Communication with the APPLICATION object are performed according to functional protocols allowing to isolated it completely from the GUI.

2.7 INFORMATION SYSTEMS

Development of Information Systems (IS) may be categorized into:

- those dealing **solely** with the definition **of a database.** In this case the use of dedicated tools (Data Base Management Systems (DBMS) development Environments, 4G languages) is required, and the use of HOOD is not relevant.
- those dealing with the definition of an application on top of a DBMS with or without development of the latter. In this case the development will obey to the separation principle of:
- the applicative part, defining the supporting architecture (interactions between the main components, possibly developed with the most suitable tools& environments),
- the MMI part (often a key part in the system),
- the interface to the DBMS, and/or the DBPMS itself.

Using HOOD such a separation is easily supported by defining an initial model comprising, a MMI object, an APPLICATION object and an DBMS_IF object which exchange data. Those data should be implemented as instances of ADTs and defined in abstract data type support objects, according to principles of section 2.1 above.

2.7.1 PARALLEL DEVELOPMENT OF INFORMATION SYSTEMS

The separation between these three kinds of components can be formally expressed with HOOD in the initial model, allows to develop in parallel:

- **the applicative part** through HOOD decomposition (see figure 5.3.2 below)
- **the MMI part according to the principles outlined above** (see section 3.5.2 above
- **the interface to the DBMS**: where several cases have to be handled:
  - **only informal descriptions upon a information / data model have been** established in requirement analysis**.** The formalization of associated data structures must be performed during the HOOD design (by means of E_R models[5], typing features of the target language and/or through HADT objects)
  - **a formal conceptual data model has been established and verified**: the model has been captured using a tool and/or notation such as Entity_Relationship (E_R) or OMT formalism. The use of appropriate tools may provide for automatic generation of software for access and management of the associated implementation model, in terms of modules, (as implementations of abstract data types[6]). Otherwise, the associated software can also be developed using HADT objects.
  - In case a relational DBMS (RDBMS) is used, the implementation of the abstract data type support HOOD object will use the SQL primitives to handle fine access to the data. In case an Object Oriented DBMS (OODBMS) is used, the abstract data type implementations may directly be the classes associated to the object oriented database schema

An example of an IS initial HOOD model is given in Figure 85 - below

[5]. E_R Entity relationships models. Modern models support inheritance to factor relationships and description, see OMT for example.

[6]. Some tools are now able to automatically generate C or C++ interface modules for manipulation of the schema and access data associated to a MERISE or E_R data models extended with inheritance

2.7.2 EXAMPLE OF A HOOD INITIAL INFORMATION SYSTEM MODEL

Three objects associated to the three development lines, and exchange data which are implemented by HADT [environment] objects.

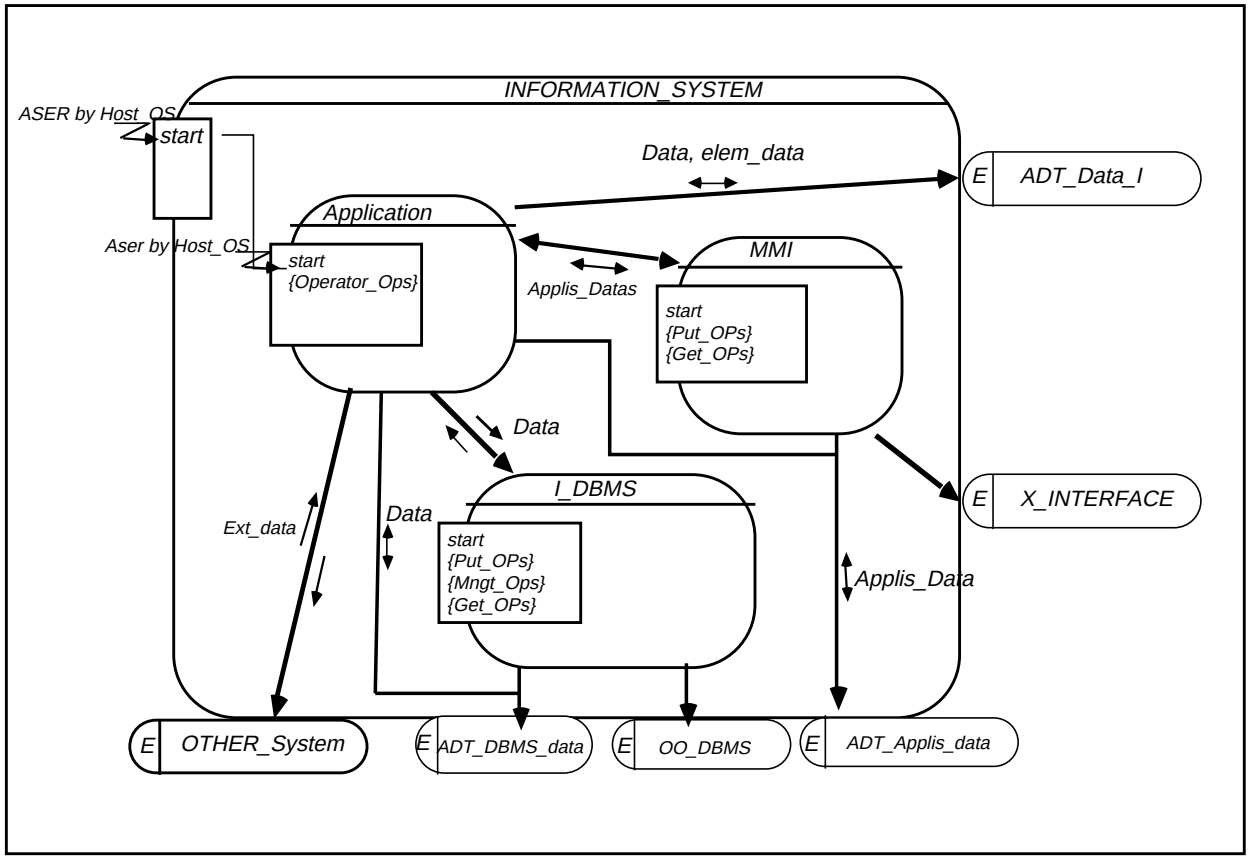


Figure 85 - Typical architecture of an Information System initial HOOD model

- **the object I_DBMS** provides a Start operation required by APPLICATION for initialisation. Operation sets {Put_Ops} et {Get_Ops} define the interactions with APPLICATION and/or MMI on readings and updating of informations stored in the DBMS.
- **the object ADT_DBMS_DATA** groups the definition of abstract data type support HOOD objects associated to data exchanges between MMI and/or APPLICATION with the DBMS.

2.8 FAULT TOLERANT SYSTEMS

In case where systems have additional fault tolerant constraints, a similar approach can be put into work :

- first, by developing *a logical solution* where the fault-tolerance constraints are not taken into account, but allowing validation of the essential properties of the solution.
- then, by refining this solution to tune, adapt ruggedizing it to handle specification or design faults, supposing that for each HOOD object, its environment is fault-tolerant.
- finally by adapting and *ruggedizing* this latter solution towards a physical solution where environment objects of all previously ruggedized objects are now considered non fault-tolerant.

2.9 ERROR AND EXCEPTIONS HANDLING

2.9.1 THE EXCEPTION CONCEPT

HOOD takes the exception concept from Ada at the design level. This concept allows to express smartly and concisely "non nominal" behaviours. However it also presents a number of drawbacks which lead us to advice very limited usage.

- First of all, the implementation of exceptions in target other than Ada sets problems of correctness and validation of the equivalent software behaviour
- Moreover, an exception describes an abnormal flow of control in reverse of the use relationship. More precisely an exception is implemented as the asynchronous execution of the exception_handler call within the client code instead of normal return after service call. Thus exception execution means asynchronous executions, that render very difficult exhaustive testing, and hence render difficult maintenance and mastering of all possible states of the software under development.

Hence we recommend:

- use only HOOD exceptions when the target language is known to be Ada,
- limit as much as possible the use of EXCEPTIONS in the design,
- always treat the exceptions locally, in order to avoid propagation and side effects of asynchronous processing.

2.9.2 ERRORS HANDLING

When developing large software, general principles of error handling must be applied and enforced, in order to standardize, modularize and reuse error handling code.
A basic principle is to handle errors locally (i.e as they are detected) since the error context is then available.
Error handling may be centralized or distributed. However for ergonomic reasons, complexity decrease, and reuse it is recommended to centralize error handling so as to have an homogeneous and standardized treatment.
Another advantage of this approach is a better maintainability, since the error handling object should group together all recoveries operations.

2.9.3 SUGGESTED SOLUTION

A good strategy for error handling could be:

- Use of one unique object in the system providing logging services for errors and exceptions and allowing to apply a minimal standard code to each error and or recovery processing. Figure 86 - below gives an example of a specialized object in the logging of error and exception handling, either within files and/or and a dedicated screen window.
- Recovery strategy on a local priority basis

.

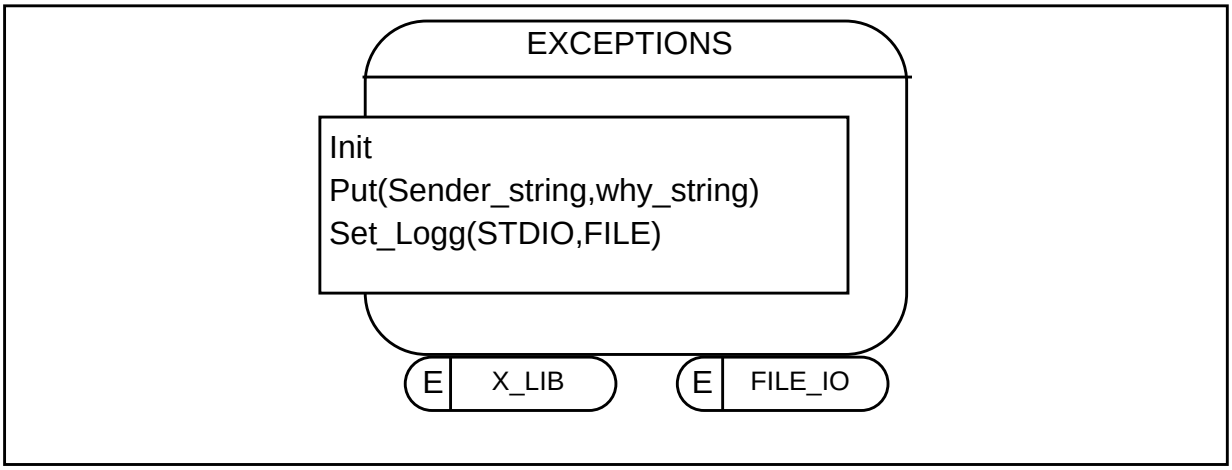


Figure 86 - Dedicated Object to handle Error and exceptions

2.10 REUSING HOOD DESIGNS

2.10.1 OVERVIEW

What is the exact meaning of reuse in our context? It can simply be stated that it is the ability to pick up parts of previous designs in order to build a new one. Some work is supposed to have already been done in an area more or less similar to those of the new project, and the problem is to get this work back rather than redoing a similar job, giving similar results at an extra charge. Two parts of the last sentences are of interest:

- "pick up parts of previous designs", this implies the possibility to easily isolate parts of a design;
- "an area more or less similar", the notion of similarity plays a extremely important role for what concerns reuse.

First, let us speak about theses parts that shall be easily separated from their original context and easily integrated in a new one. This is one of the definitions of "object": a part of a software clearly delimited which provides clear interfaces with the external world. The HOOD object orientation is the starting point of the whole reuse methodology.
Second, what about similarity? It does not mean equality, hence it seems clear that even if a certain number of objects may be picked up directly and reused as they are, it is not the general case, and some others should be Adapted to their new context. There is mainly two different techniques to achieve this purpose:

- genericity: the object to be reused is generalized and transformed in a template which may be instantiated for each specific context. This technique is covered by the Ada generics, the C++ templates, the Eiffel parameterized classes and the HOOD3 Classes.
- inheritance: the object to be reused does not change, but the new Adapted object is defined by means of enhancement of the former. It automatically inherits all its properties and operations, one has just to redefine the methods which differ from the original. This very powerful mechanism is known as "inheritance" and is supported by most Object-Oriented programming languages.

As the HOOD3 method does not support direct inheritance, only the first method will be presented in this part. The reader shall be aware of the existence and the importance of the other technique which is known as the best reuse support technique at the programming level. Studies in this direction are in progress in the context of ESPRIT/PROTEUS project, to allow the mapping into C++, a language providing inheritance mechanisms (see part II of this document).

HOOD means Hierarchical Object-Oriented Design. We presented Object-Orientation as a key point about the reusability. Curiously, the name of the method, in itself, contains all its advantages and its drawbacks. From the reuse point of view, the Hierarchical aspect of the method is an important hindering. The objects are distributed all over the design tree, and one may be led to reuse a complete sub-tree whether a linear design would have led to reuse just a couple of interrelated objects. Furthermore if a non-terminal object has to be Adapted, the modifications will be propagated to lower-level objects, and the repercussions may be difficult to evaluate.

In order to achieve this goal, one must be aware of the fact that it is quite impossible to design from scratch a perfect reusable object. The first version is often a draft, and further versions follow with improvements and modifications. A real problem occurs with all the design documents:

is it possible, desirable to rewrite all the stuff at each new version? This document will show that there are different kinds of designs, some of them are likely to change rapidly, some others may be stabilized more rapidly. The first category does not need to be to much well-documented although the second one is more ready to this work.

2.10.2 TOP-DOWN & BOTTOM-UP APPROACHES

Traditional software development methods enforce a top-down approach (see figure 87). This is quite a natural way of thinking and has big advantages for building complex systems: it gives a method for breaking down a system in smaller parts easier to tackle directly. When those parts are still too big to be treated, they can be broken down again. Finally, this is a good way of building rapidly a solution of a given problem.

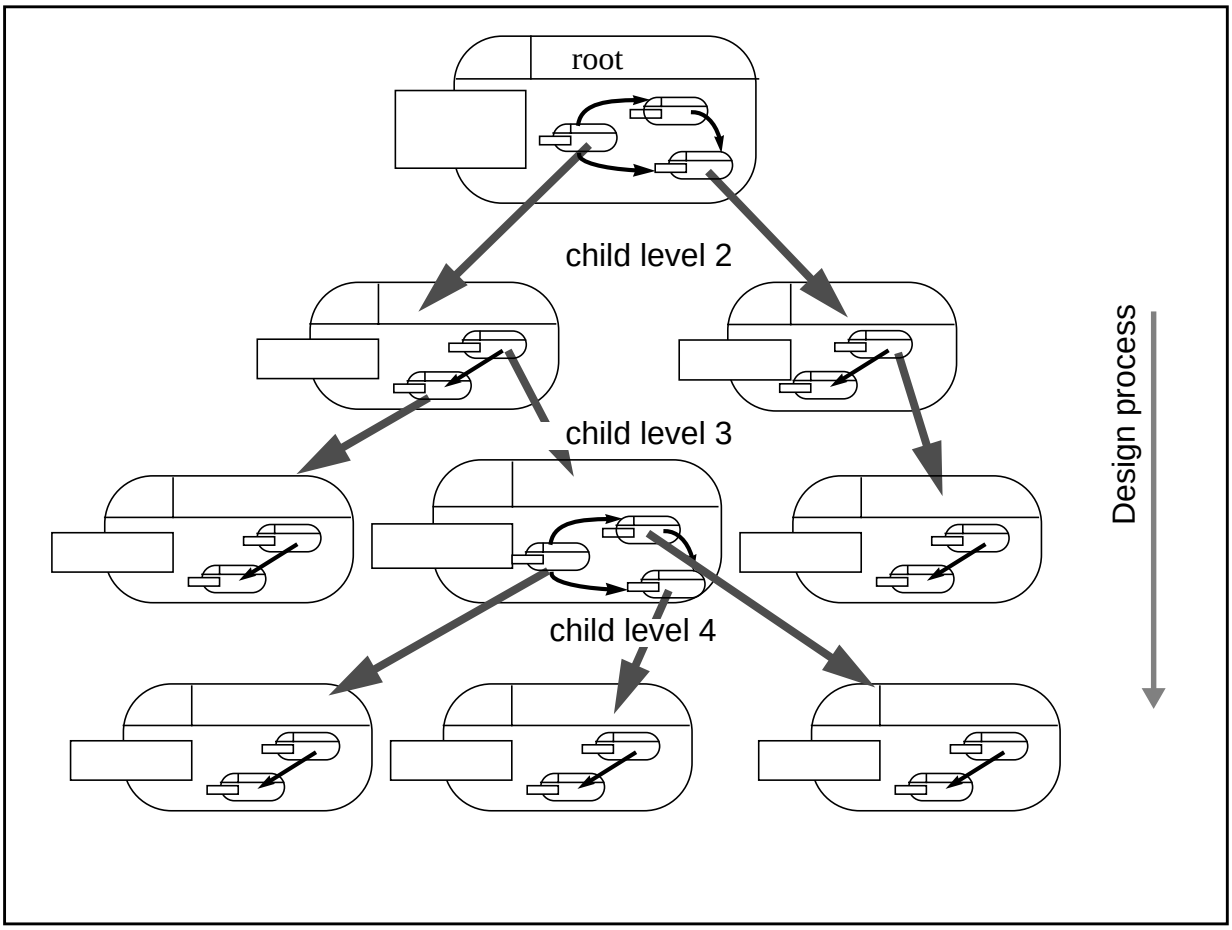


Figure 87 - Classical Top-Down Approach

New development methods based on the object-oriented technology tend to renounce to this type of approach and insist more on bottom-up processes (see figure 88). It consists in building simple components, which are supposed to simplify problem definition, and then building more sophisticated components on the top of them, until the solution appears as a trivial application of the ultimately developed tools

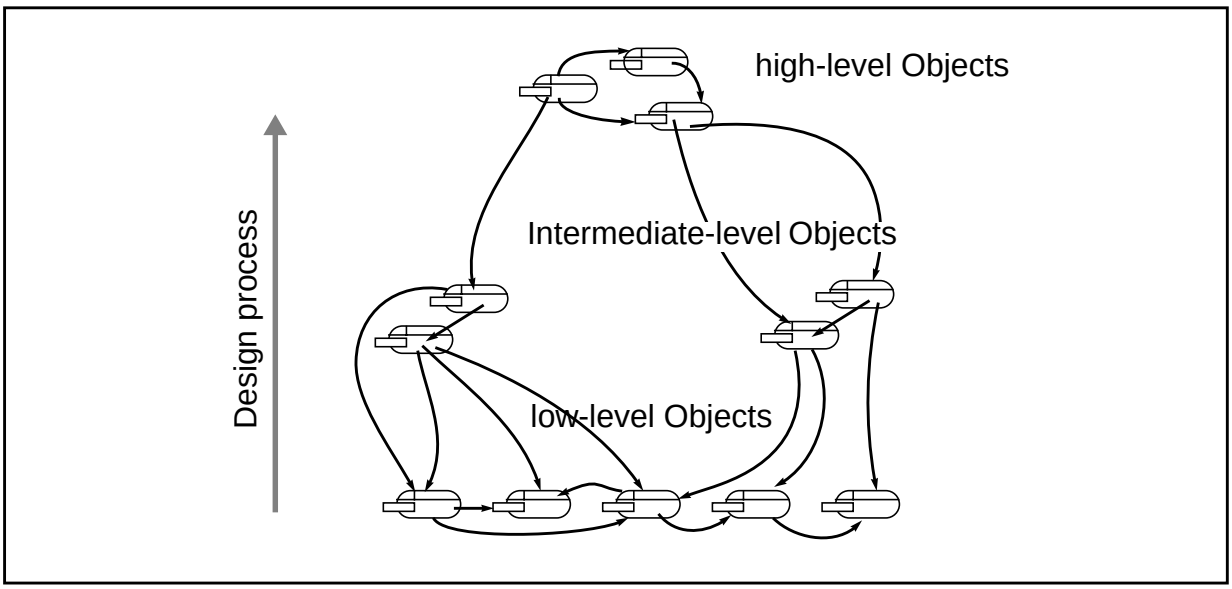


Figure 88 - Layered Bottom-Up Approach

At a first glance, the bottom-up method may look like risky, since nobody can insure that this process actually leads to a correct solution. A great amount of time may be spent to develop complex components that will never or very partially be used during the current project. On the other hand, as components are developed more or less independently of the current project, they are more naturally reusable for new projects. Hence, after a starting period, a great number of low-level objects can be picked up from related projects formerly developed. As far as we are concerned, this point is very important.
Another point is to be mentioned. The top-down approach has been presented as the fastest way to elaborate the architecture of a system, provided that clear and unambiguous specifications of the problem have been given. Unfortunately this is almost never the case, as specifications are often not entirely clarified before the delivery of the project. The common case shows the client needs evolving during the development process, even during the maintenance phase. Within a top-down approach, a small modification in the specification of a high-level object may induce a complete redesign of the internal objects.
As we have seen that both approaches have advantages and drawbacks, we suggest an intermediate path between these two opposite ways of thinking (see figure 89). Two activities can be led in parallel: the first one is in charge of building the architectural design in a traditional top-down manner, while the second one is in charge of the construction or enhancement of reusable libraries.
The first activity shall not overcome a depth of 2 (or 3 for big projects) levels of decomposition. Every object produced in this phase has a good chance to become a non-terminal object. Thus it can be considered as a good design practice to design them as simple abstract objects". For describing these objects, emphasis shall be put on the H2 ("Informal Solution Strategy") and H3 ("Formalization of the strategy") chapters of the HOOD Chapter Skeleton. The HOOD informal graphical description takes all its sense at this level.
The second activity can lead to the creation of a lot of terminal objects. It is important, for the feasibility of this phase, to rely on a good managing tool for libraries of reusable components. However some possibilities of structuring the reusable objects by grouping them into non-terminal objects will be explained later. This technique allows designers to organize a reuse activity at different level of granularity.

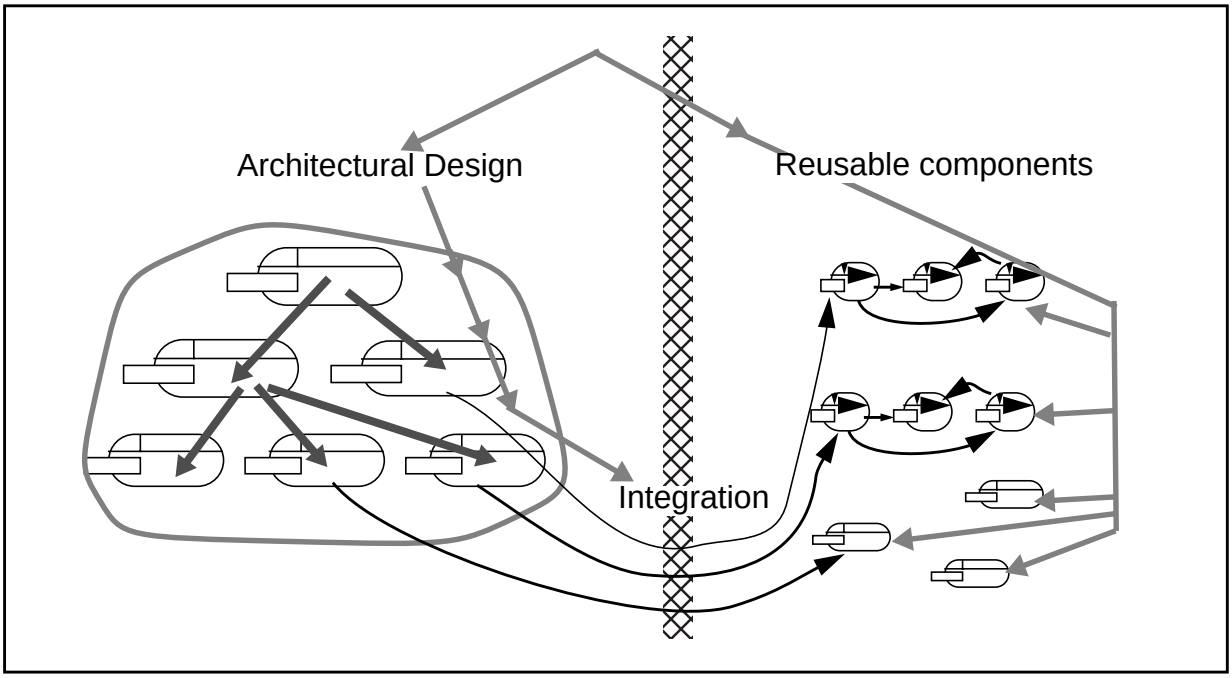


Figure 89 - Mixed Approach

This mixed approach imposes some constraints in the development process.
First of all, the previous phases (requirements, analysis) have to be modified in order to provide two types of information: as usual, the general requirements of the system as a whole in order to achieve the architectural design; but also, another analysis of these requirements in order to identify the low-level objects constituting the environment of the system.
Note that the integration step can be tricky. It is recommended that the architectural design team has a correct idea of what it can get from the reusable libraries, and on the other hand, even through reusable components should be created as independent as possible, it is preferable that the team in charge of it has a correct understanding of the context where they will be used.

2.10.3 GENERAL GUIDELINES

This section gives general guidelines for producing “good” HOOD design, i.e. designs encouraging reuse and making evolution easier.
It is a hard work to give a complete list of all the advice that can help a designer in charge of building a highly reusable and evolutionary software. It is even harder to find a good way of presenting it. One can either provide very general rules that give the spirit of design with reuse, or offer a plethora of very precise guidelines, well-documented and easy to understand. The main drawback of the first solution is its difficulty to be applied to a particular case. The last solution implies some experience to know where and when to apply a precise guideline and a good memory to remember so many rules.
As these two ways form a well balanced whole, both are presented. In a first part, the very few fundamental rules are expressed. They are taken up again, completed, detailed and exemplified in a second part.
These guidelines are provided with the following template:

- advice: an advice number, e.g. *advice i* and a short sentence summarizing the advice

- comments: detailed comments on the advice, including an explanation of why giving the advice, details on how to apply it...
- example: if necessary, an example illustrating the advice

2.10.3.1 General design techniques

This section contains the fundamental statements designers must have in mind during all the design process.

Advice 1 Design objects as independent as possible from their context.

Comments: While creating a design, designers have in mind the specification of the problem to be solved. One of the most common un-reusable and anti-evolutionary practice consists in defining objects with operations that are not part of their own functionalities but those of the system itself. Objects must be understood as servers or tools that help solving the problem but that should not solve a part of the problem by themselves.

Example: Suppose that we want to count the occurrence of each word in a text. Objects as Text, Word and Occurrence model the problem space. Then, we need an object Map in order to associate the words with their running occurrence. The good design practice is to design (or to reuse) a general map that associates the keys (here words), with their values (here words occurrences). Operations can be, for instance, `Add_a_new_key`, `Change_the_value_of_a_key`, `Retreive_a_value_from_a_key.` A less correct design solution would have been to define a specific map that automatically increments the value, each time the key is used with operations like `Add_a_new_key`, `Increment_the_value_of_a_key`. It would have been interesting (but not so much) in the context of this particular problem but would have lost its generality, and then the opportunity to be reused elsewhere. Furthermore the "`increment`" operation is quite easy to implement by a simple expression of the general set of operations: `Change_the_value_of_a_key` by `the value Retrieved from_the_same_key` incremented by one.

Advice 2 Do not over-specify objects

Comments: In the opposite, a well-intended designer may want to create real reusable objects. He can try to imagine every context where the object could be reused, find exceptional cases and try to solve them, although it has nothing to do with his current system design. This is a risk to spend too much time and energy on this object forgetting actual purposes, all the system design being delayed for non-tangible future advantages. Furthermore, the object being created has good chances to become a kind of monster, willing to do everything but doing nothing correctly.

Advice 3 Use Simple Abstract Objects for higher-level objects and active objects.

Comments: This is detailed in advice 27. The next section exemplified the fact that high-level objects fit often better with Simple Abstract Objects.

Advice 4 Prefer Abstract Data Types for passive terminal objects.

Comments: This is detailed in advice 26.

Advice 5 Take care of the direction of the USE relationship between objects.

Comments: There is often a design choice consisting in deciding when two objects need to communicate, which one will be the client and which one will be the server. It may be a difficult choice and this decision has to be taken carefully (see advice 20 and advice 21).

Advice 6 *Do not use more than three or four levels of INCLUDE relationships in the architectural design.*

Comments: In general, two till three levels is enough for the architectural design when applying the mixed bottom-up & top-down approach previously presented.

Advice 7 *Define the most obvious reusable parts as independent designs or classes.*

Comments: The mixed bottom-up & top-down approach leads to split designs where reusable parts are designed independently, allowing a relatively low number of levels, as said in advice 6.

2.10.3.2 Classified guidelines for reuse and evolution

This section presents advice for professional HOOD designers. This advice is intended to enforce the potential reuse of the object being designed. It is grouped by themes:

- understanding and clarity: how to follow a good naming strategy.
- robustness: how to design objects that can be used even under unexpected conditions.
- Adaptability: how to design objects so that evolutions are easy and as localized as possible.

Some of the following advice is inspired by a guide for Professional Ada Programmers and Adapted to the HOOD context.

2.10.3.2.a Understanding and Clarity

Advice 8 *Select the least restrictive names possible for the reusable objects and their identifiers*

Comments: nobody really knows the context of future reuse.

Advice 9 *Reserve the best name for the Instance Object and the second best name for the Class itself*

Comments: When there is an obvious choice for a simple and clear name for the reusable object, it is a good idea to keep this name for the instance, choosing a longer, more descriptive name for the class. Thus, `GENERIC_STACK` is a better name than `STACK` for the class, because it leaves the simplest name `STACK` available to be used for an instantiation.

Advice 10 *Do not use any abbreviations in reusable objects and their identifiers.*

Comments: If the object is really reusable, nobody can ensure that an abbreviation makes sense in any context of future reuse.

2.10.3.2.b Robustness

Advice 11 *Use symbolic constants to allow multiple dependencies to be linked to a small number of symbols.*

Comments: The evolution process often implies small changes in dependencies. The possibility to map the new dependencies just by changing few constants greatly improves the ease of reuse and greatly reduces the number of opportunities for errors.

Example: Figure 90 - is an example of using constants. Some of the symbols, those which are likely to evolve and thus have to be changed in future systems may be declared and defined in the provided interface. The other ones may only be declared in the provided interface and defined in the internals. HOOD gives this possibility to enforce information hiding at the design level.

When implementing such a design, you may or may not have a direct mapping depending on the target language you use. For example, Ada gives the possibility to define private types (even if it forces the definition to be in the specification part). Other target languages (such as C, pascal, Fortran,...)do not offer this possibility. But in any case, the principle remains the same at design level.

```
OBJECT Disk_Driver is ACTIVE
 DESCRIPTION
    --| description |--
 IMPLEMENTATION_CONSTRAINTS
    --| constraints |--
 PROVIDED_INTERFACE
    TYPES
       SectorRange ;
       TrackRange ;
       SurfaceRange ;
       Track_Map         is array (Sector_Range)  of ...;
       Surface_Map       is array (Track_Range)   of Track_Map;
       Disk_Map          is array (Surface_Range) of Surface_Map;
    CONSTANTS
       Number_Sectors     : constant Integer := 4;
       Number_Tracks      : constant Integer := 200;
       Number_Surfaces    : constant Integer := 18;
       Sector_Capacity    : constant Integer := 4096;
       Track_Capacity;
       Surface_Capacity;
       Disk_Capacity;
...
 INTERNALS
    TYPES
       SectorRange    is range 1 .. Number_Sectors ;
       TrackRange     is range 1 .. Number_Tracks ;
       SurfaceRange   is range 1 .. Number_Surfaces ;
    CONSTANTS
       Track_Capacity   : constant := Number_Sectors * Sector_Capacity ;
       Surface_Capacity : constant := Number_Tracks * Track_Capacity ;
       Disk_Capacity    : constant := Number_Surfaces * Surface_Capacity ;
```

provided & hidden types

provided & visible types

provided & visible constants

provided & hidden constants

Figure 90 - Definition of a disk driver and its associate properties

Advice 12 Use unconstrained array types for formal parameters and array return value.

Advice 13 Make the size of the local variables depend on actual parameter size where appropriate.

Comments: This is the best way of defining reusable operations on arrays. The local variables are correctly dimensioned in any case.

Example: Figure 91 - gives an example of defining an array.

```
INTERNALS
 TYPES
    Vector  is array (Index range <>)                 of Element ;
    Matrix  is array (Index range <>, Index range <>) of Element ;
 OPERATION_CONTROL_STRUCTURES
    OPERATION Matrix_Op(A : in Matrix) IS
       ...
       CODE
          Workspace : Matrix(A'RANGE(1), A'RANGE(2));
          TEMP_V    : Vector(A'FIRST(1) .. 2 * A'Last(1));
       ...
```

Figure 91 - defining an array

Advice 14 Be careful about overloading the names of provided operations in a HOOD Class

Example: Figure 92 - is an example of an INPUT_OUTPUT class. If this class is instantiated in a IO_INTEGER object (see figure 93), the two Put operations have identical interfaces, and all calls to Put are ambiguous. Therefore this class cannot be used with the type Integer (and its sub-types).

```
OBJECT INPUT_OUTPUT is CLASS PASSIVE
 FORMAL_PARAMETERS
    TYPES
       Items is limited private;
 PROVIDED_INTERFACE
    OPERATIONS
       Put(Item : in Integer);
       Put(Item : in Items);
```

Figure 92 - two different put operations

```
  OBJECT IO_INTEGER is INSTANCE_OF INPUT_OUTPUT
   PARAMETERS
      TYPES
         Items => Integer;
   ...
```

Figure 93 - instantiation of `INPUT-OUTPUT`

In such a case, it is better to give unambiguous names to all operations.

Advice 15 Use exceptions carefully. Use exceptions only for uncontrolled errors and handle anticipated errors specifically in the object.

Comments: There are problems in using freely exceptions. The first point is that using exceptions is not fully compliant with the encapsulation principle. On another hand, using exceptions may lead to problems while coding, depending on Ada comSTACKrs. There are two types of error processing: uncontrolled and anticipated. Uncontrolled errors may be managed by using exceptions. This include Ada predefined exceptions. Uncontrolled means errors can occur at any moment and are not specific to an object. Anticipated exceptions are for instance the example of

the stack with the `is_empty` operation; they are “part of” the object. All which is specific to an object has to be treated as an error processing specifically in the object, otherwise use (carefully) exceptions.

Advice 16 *Propagate exceptions out of reusable parts. Handle exceptions internally only when you are certain that the handling is appropriate in any circumstance.*

Advice 17 *Always propagate exceptions raised by class formal operations.*

Advice 18 *Always come back to the last correct state of the object when raising an exception.*

Advice 19 *Always leave parameters unmodified when raising an exception.*

Comments: When designing a reusable component, exception handling must be done carefully and seriously. It is generally not possible to handle exceptions internally, because the environment may vary a lot from one case to another, especially when it occurs during the evaluation of a formal operation. So exceptions have to be propagated. But before propagating the exceptions, designers must be sure that the object is left in a correct state.*Figure 94 -* shows how to handle an exception in order the regenerate a correct state before propagating the error.

```
OBJECT PRINTABLE_STACK is CLASS PASSIVE
   FORMAL_PARAMETERS
      TYPES
         Items is limited private;
      CONSTANTS
         SIZE: Integer;
      OPERATIONS
         Put   (Item: in Items);
   PROVIDED_INTERFACE
      OPERATIONS
         Push (Item: in Items);
         Pop   (Item: out Items);
         Print;
      EXCEPTIONS
         X_Stack_full;
   INTERNALS
      TYPES
         Stack_Range is Integer range 1.. SIZE;
         Stack is array(Integer range <>) of Items;
      DATA
         Top: Integer := 0 ;
         The_Stack : Stack(Stack_Range) ;
      OPERATION_CONTROL_STRUCTURES
         OPERATION Print is
         ...
      PROPAGATED_EXCEPTIONS
         others ; --| all the exceptions raised during the formal operation "Put" are propagated |--
         CODE
            begin
            ...
            begin
                  Put(The_Stack(i)) ;
               exception
               when others ==>
                  COME_BACK_TO_A_CORRECT_STATE;
                  raise ;  -- propagation outside
            end ;
```

Figure 94 - A generic stack providing a print operation

2.10.3.2.c Adaptability

Advice 20 Minimize the number of objects a reusable object uses.

Comments: A “good” reusable (stable) object is used by a lot of other objects but uses as few other objects as possible. No object used is perfect but only applies to a limited type of objects.

Advice 21 Minimize the number of objects using objects that may vary.

Comments: this is similar to the previous advice. This is related to the principle of low coupling, enforced by the notion of variabilities. The objective is to minimize the impact of changes propagating following the USE relationship, by minimising the USE relationships to objects which have been identified as potentially variant.

Example: a simulator simulates or is connected to a certain number of satellite sub-systems of different types (for instance power subsystem, attitude and orbit control subsystem,...). One to N specific objects of this type may be present in a Simulation object. In that case, it is good to minimize the number of objects using this kind of Sub-systems objects; this avoids changing the Required Interface in the object(s) using them.

Advice 22 Provide complete functionality in objects. Provide initialization, reset and termination operations when appropriate.

Comments: This is particularly important when designing an abstraction. The essence of Object-Based Design is the design of complete abstraction even if it is not required in the current application. When possible, an abstraction should be initialized automatically. Probing operations may be provided to determine limit cases, so that the user can avoid causing exceptions to be raised.

Advice 23 Use HOOD Classes to avoid code duplication

Advice 24 Parameterize Classes for maximum Adaptability

Advice 25 Use Classes to encapsulate algorithms independently of data type.

Example: Figure 95 - is an example of a class representing an object to sort data independently of data type.

```
OBJECT SORT_SERVER is CLASS PASSIVE
 FORMAL_PARAMETERS
    TYPES
       Element is limited private ;
       Data is array(Integer range <>) of Element;
    OPERATIONS
       "<"(Left : in Element; Right : in Element) return Boolean is <>;
       Swap(Left : in out Element; Right : in out Element) ;
 PROVIDED_INTERFACE
    OPERATIONS
       Sort(Data_to_Sort : in out Data);
    INTERNALS
       OPERATION_CONTROL_STRUCTURES
       OPERATION Sort(Data_to_Sort : in out Data) IS
       ...
       CODE
       begin
       ...
       for I in Data_to_Sort'range loop
       ...
           if Data_to_Sort(I) < Data_to_Sort(J) then
              --| use of formal "<" |--
              swap(Data_to_Sort(I), Data_to_Sort(J)) ;
              --| use of formal "swap" |--
           end if ;
       ...
       end loop ;
```

Figure 95 - an object to sort data

There can be different instances: an instance for sorting integers (see figure 96),

```
OBJECT SORT_INTEGER is INSTANCE_OF SORT_SERVER
 PARAMETERS
    TYPES
       Element => Integer ;
       Data    => UTIL_INTEGER.Table;
    OPERATIONS
       Swap    => UTIL_INTEGER.Swap ;
 ...
```

Figure 96 - Instance for sorting integer

an instance for sorting strings (see figure 97),...

```
  OBJECT SORT_STRING is INSTANCE_OF SORT_SERVER
   PARAMETERS
      TYPES
         Element => UTIL_STRING.Line ;
         Data    => UTIL_STRING.Table_of_Line;
      OPERATIONS
         Swap    => UTIL_STRING.Swap_Line;
   ...
```

Figure 97 - Instance for sorting string

Advice 26 For passive objects, use Abstract Data Types in preference to Abstract Data Ob-

jects.

Comments: The biggest advantage of an ADT over an ADO is that the user can declare as many objects as desired with an ADT. These objects can be used as part of other objects: they can be declared as components of arrays or records. They can be used to instantiate classes with formal type parameters. Finally, this is the only way to cope with a dynamic number of objects.

Advice 27 For active objects, use Abstract Data Objects in preference to Abstract Data Types.

Comments: The problem with ADT in HOOD is that there is only one “HOOD Object” defined and visible in the design and maybe a lot of “real objects” instantiated from this ADT, hidden in the internal part of terminal HOOD objects. Since the object is passive, this is not a real problem, but as soon as the object becomes active it is not recommended. The Ada language allows the building of correct concurrent ADT, but it is not a simple matter. Booch has proposed different models for concurrent ADT in the famous[GON 90]: the guarded form, the concurrent form and the multiple form. [GON 90] has proved that at least the concurrent form was erroneous because it was depending on the parameter passing semantics which can differ from a compiler to another. Finally [CAR 91] proposed a seemingly correct form for concurrent ADTs. This form relies on a Task type hidden by a private type definition. Even if this model seems to be correct and satisfactory, it is not easy to use it with the HOOD formalism. With HOOD, the principle of active objects is that the behaviour is described within the OBCS and typically the OBCS is one task. In any case, as explained in [CAR 90]the use of Abstract State Machine (i.e. Abstract Data Object) is a good software engineering practice for concurrent components.

Advice 28 Use HOOD Classes to implement Abstract Data Types and Abstract Data Objects independently of their component data type.

Comments: this is the standard way of generalizing ADT and ADO.

Example: The example of the bounded stack shows different means to represent the size constraint of a bounded structure.

- a provided constant: in `BOUNDED_STACK` and `BOUNDED_STACKS`, `MAX_STACK_SIZE` is a provided constant used in internals as the size of the array implementing the stack.
- a constant formal parameter: in `BOUNDED_GENERIC_STACK` (see figure 98), `MAX_STACK_SIZE` is a formal constant parameter used in the same way.

```
OBJECT BOUNDED_GENERIC_STACK is CLASS ACTIVE
  DESCRIPTION
     --| a protected stack as an abstract data object |--
  FORMAL_PARAMETERS
     TYPES
        Elements is limited private;
     CONSTANTS
        MAX_STACK_SIZE : NATURAL  := 100;
     OPERATIONS?
        Assign(From : in Elements; To : out Elements);
  PROVIDED_INTERFACE
     OPERATIONS
        Push(Item : in  Elements) ;
        Pop (Item : out Elements) ;
     EXCEPTIONS
        X_OverFlow ;
        X_UnderFlow ;
```

Figure 98 - Generalization of the simple ADO BOUNDED_STACK

- a discriminant of a private type: in BOUNDED_GENERIC_STACKS (see figure 99), MAX_STACK_SIZE is a discriminant used for constraining the array component of the internal stack record.

```
OBJECT BOUNDED_GENERIC_STACKS is CLASS PASSIVE
  DESCRIPTION
     --| a simple generic stack as an abstract data type |--
  FORMAL_PARAMETERS
     TYPES
        Elements is limited private;
     OPERATIONS
        Assign(From : in Elements; To : out Elements);
  PROVIDED_INTERFACE
     TYPES
        Stacks(MAX_STACK_SIZE: Natural := 100) is limited private ;
     OPERATIONS
        Push(Stack : in out Stacks; Item : in  Elements) ;
        Pop (Stack : in out Stacks; Item : out Elements) ;
     EXCEPTIONS
        OverFlow ;
        UnderFlow ;
  INTERNALS
     TYPES
        Stack_Body  is array(Positive range <>) of Items;
        Stacks(MAX_STACK_SIZE: Natural := 100) is record
            Top  : Natural := 0;
            Body : Stack_Body(1..MAX_STACK_SIZE);
        end record;
```

Figure 99 - Generalization of the simple ADT BOUNDED_STACKS

These three cases imply to choose the actual size of the structure at different moments.

- in the first case, the decision is taken when designing the object itself. Changing this value means changing the object and it may have important consequences (re-code generation, re-compilation, re-validation etc.).
- in the second case, the decision is taken when instantiating the class.The class definition has

not to be modified.

- in the third case, the decision is taken when designing the object using the stack, for examples by declaring a value of the provided type. In particular, the actual size may only be known at run-time, the same instance of the class is able to produce values of different sizes.

Advice 29 *Use limited private types (not only private) for Class formal types parameters, explicitly importing assignment and equality operations if required.*

Comments: For a class to be usable in as many contexts as possible, it should minimize the assumptions that it makes about its environment and should make explicit any assumptions that are necessary. A formal limited private type prevents the class from making any assumptions about the structure of objects of the type, or about operations defined for such objects. A private formal type (non limited) allows the assumption that assignment and equality are defined for the type. Thus, a limited private type can not be specified as the actual parameter for a private formal type.

Example: Figure 100 - is an example of what not to do, whereas *Figure 101 -* is an illustration of the advice.

```
OBJECT H_LIST is CLASS PASSIVE
 TYPES
    Items is private;
    Keys  is private;
    OPERATIONS
       Key_Of(Item : in Items) return Keys;
 PROVIDED_INTERFACE
    TYPES
       Lists is limited private ;
    OPERATIONS
       Insert  (List : in out Lists; Item : in  Items);
       Retrieve(List:inoutLists;Key:inKeys;Item:inoutItems);
    ....
```

Figure 100 - A class violating Advice 29

```
OBJECT H_LIST is CLASS PASSIVE
 TYPES
    Items is limited private;
    Keys  is limited private;
 OPERATIONS
    Key_Of(Item : in Items) return Keys;
    Assign(From : in Items; To    : in out Items);
    "="   (Left : in Keys ; Right : in      Keys) return boolean;
 PROVIDED_INTERFACE
    TYPES
       Lists is limited private ;
    OPERATIONS
       Insert  (List : in out Lists; Item : in  Items);
       Retrieve(List : in out Lists; Key : in Keys; Item : in out Items)
  ....
```

Figure 101 - The same class respecting Advice 29

Note that the first example implicitly defines equality and assignment for both formal types `items` and `keys` even though the second example only defines assignment for `items` and equality for `keys` which is sufficient. Although the first example seems shorter, it makes too much assumptions on its formal parameters and is then less reusable.

2.10.4 ADVANCED TECHNIQUES

This section contains details on HOOD specific advanced techniques which are to be known by designers to build reusable and easily evolving designs. Among these techniques are the use of classes and virtual nodes.

2.10.4.1 Transforming objects into HOOD3 classes

Coupling between objects has to be avoided as often as possible (principle of weak coupling), and especially for reusable objects. A potential user is less ready to reuse an object if it requires the use of other parts which seem unnecessary. This “extra luggage” wastes time and space. Coupling between reusable parts should only occur when it provides a strong benefit perceptible to the user.
A technique to postpone coupling is now presented. When a reusable object needs services from some other ones, it is possible to transform the direct dependency (required interface) into generic parameters (class formal parameters).

```
OBJECT EXEMPLE is PASSIVE
  ...
  REQUIRED_INTERFACE
     OBJECT LIST
     TYPES
        Lists;
     OPERATIONS
        Insert(List : in out Lists; Item : in  Elements);
        Remove(List : in out Lists; Item : out Elements);
```

Figure 102 - an Object with explicit dependency

As shown in *Figure 103* - , it is sufficient to move the required constants, types and operations from the required interface (see figure 102) to the formal parameters(see figure 104).

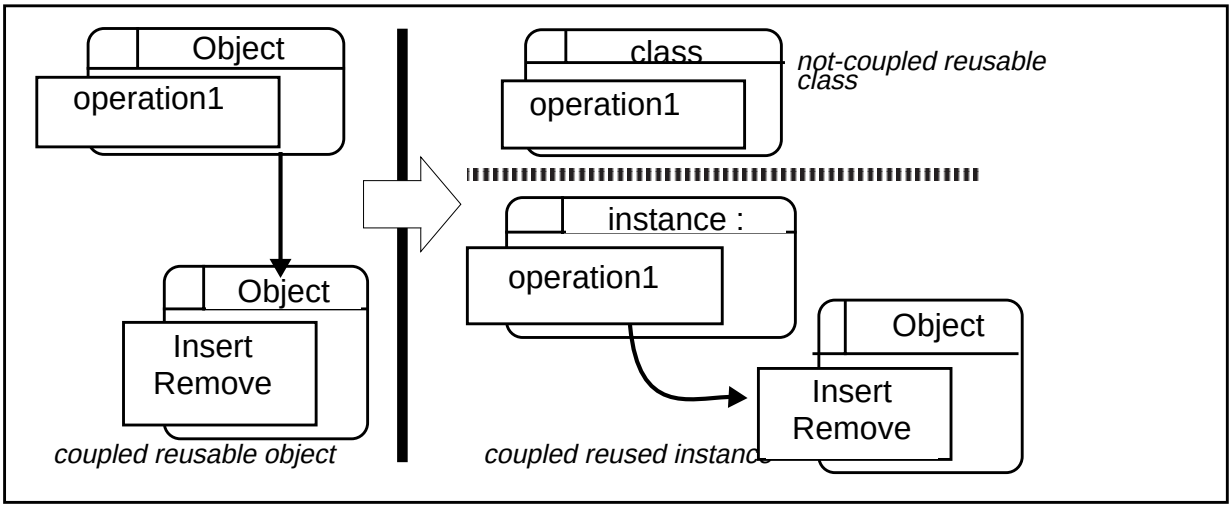


Figure 103 - de-coupling a reusable object

```
OBJECT EXEMPLE is CLASS PASSIVE
  ...
  FORMAL_PARAMETERS
     TYPES
        Lists is limited private;
     OPERATIONS
         Insert(List : in out Lists; Item : in  Elements);
        Remove(List : in out Lists; Item : out Elements);
  REQUIRED_INTERFACE
     NONE
  INTERNALS
     --| Exactly the same a the coupled Objects unless that the dotted notation
shall not be used with Insert & Remove operations |--
```

Figure 104 - Transformation of the above Example into a class

This is not possible in any situation due to constraints on formal parameter format. But this kind of practice is very easy when the required objects have an interface corresponding to standard ADOs and ADTs (only private provided types and operations on them).
This technique, when correctly applied gives a lot of benefits. For instance, when the reusable components library contains different forms of the same reusable object (bounded and unbounded, protected and unprotected, managed and unmanaged list) you are not obliged to bind your new reusable object with one particular form, the choice can be delayed until the reuse stage. Moreover, it is a good way to reduce dependency between objects enabling to reuse in different and unpredictable contexts: in the above example, only one insert and one remove operations were required on a certain abstract type, thus whatever object providing this functionality may be used to instantiate the class even when it has been designed for something absolutely different.

2.10.4.2 Using Virtual nodes

A feature of HOOD which should be more used is the concept of virtual node.
A virtual node is a collection of HOOD objects defining a unit of distribution. This unit of distribution is allocated later onto a physical node. The activities of a virtual node consists of local operations and communications as well as remote ones with other virtual nodes.

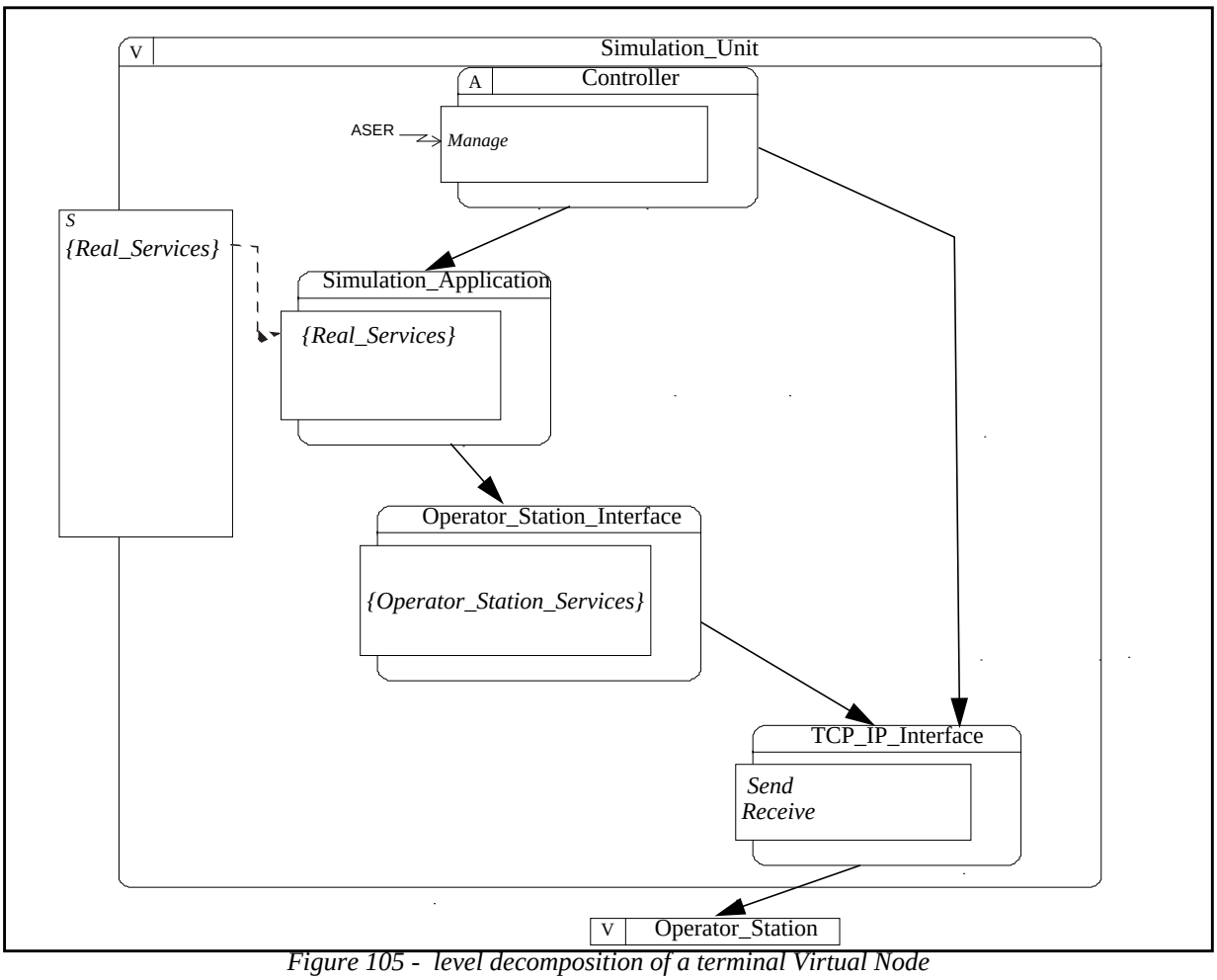


Figure 105 - level decomposition of a terminal Virtual Node

A distributed application is structured into virtual nodes accordingly to HOOD model. Thus virtual nodes are defined in a Virtual Node Tree where terminal virtual nodes are defined by allocating HOOD objects from different HOOD Design Trees.
The interest of virtual nodes resides in the fact that they free the designer of the exact hardware configuration while designing software. This very useful for reuse and evolution since the logical architecture is independent of distribution choices.
Within a terminal virtual node, it is recommended to identify the different types of logical objects and to separate them in separate levels:

- the applicative level contains objects defined according to functional requirements. They are independent of physical implementation and communication means
- the basic level contains objects dealing with physical implementation and communication means, independently of the application
- the intermediate levels contain objects allowing the two previous levels to communicate

An example of such an organisation is given in *Figure 105 - *. The applicative level is isolated in `Simulation_Application` which implements real services. `TCP_IP_Interface` belongs to the basic levels and provides communication with the virtual node `Operator_Station`. `Operator_Station_Interface` is a virtual server object of the intermediate level. `Controller` manage the application and receive messages from `TCP_IP_Interface`.

2.10.5 SUMMARY ON HOOD & REUSE

As a conclusion to this section, we shall summarize some of our conclusions based on our experience one reuse, and addressing the most important questions when wanting to go reuse:

- what is reusable?
- how do reusable?

2.10.5.1 What to Reuse?

From our experience, we have identified the following kind of reusable entities:

- **applicative domain entities,** where a number of reusable entities can be found, but their reuse is never straight forward. For a domain like *compilation*, one might for example find a simple lexical analyser up-to the full Ada compiler, but a reuser will never find the right component just suited to its last syntactic analysis problem. Briefly speaking, and according to the domain, the reuser may often be in trouble for choosing some component:
  - the one very suitable, but very difficult to Adapt in the target environment,
  - the one possibly suitable, but whose properties are hard to find or not available,
  - the one that his colleague has already flagged as *bugged*
  - the one which is to be fully completed, but also a heavy piece for the job,
  - the one easy to understand, but not flexible to support any extension
- **utilities** and small programming utilities (s*uch as data structure management modules, I/O handling modules, iterator handling modules -possibly implementing abstract data types-)* which are always present in most software. These are components which can be fully mastered by the reuser and should provide for merely direct selection, thanks to their low complexity. However, (at least in the beginning of reuse) due to previous “reuse-blind-development”, these components (*when not properly designed as implementation of abstract data types)* require still too often a real Adaptation effort to make them work in their new use environment.
- Lessons learned here are that **for a component to be reusable, it should be designed as an implementation of an ADT (Abstract Data Type)**, and possibly built on top of other ADT implementations, as the only criteria which reduces significantly Adaptation effort, and at the same time increasing testability.
- **required environment entities (Documentation, tests, test sets**). Besides the source code, it appeared as important to be able to get access to a component required environment in order to allow Adaptation. Although we could only get code components in our experimentation, we think this feature as most necessary for making reuse effective.

2.10.5.2 How to Do Reusable?

Practising reuse highlights problems related to:

- understandibility of potential components
- tuning work of a selected component within its new working environment

It seems that **reusable components** (and even reusable structures of any type) **should have three fundamental and orthogonal properties**:

- **modularity:** components should be self-documented and "self-handable" entities whose dependencies should be limited and/or systematically stated. As such, the concepts of object orientation enforcing abstraction and encapsulation of *provided interface*, as well as specification of *required interface* are of primary importance. Their application can lead to better understandibility of modules, hiding complexity and where consistency is still verifiable through controlled dependency and visibility rules. Especially these rules should b**anish any implicit visibility** what would increase the understanding effort.
- In any case the associated documentation must be "self content" and shall not have any forward or aside references.
- **invariance** which characterises the ability of a structure to resemble an ideal one or/and a standard one. It is easy to imagine that for an application domain where systems with similar functionalities are developed, there exist, in theory, an ideal organisation and structures that performs the best trends achieving the best properties. Such an architecture and associated definition of its components can be progressively refined through capitalisation of the design experience when the system is refined version after version. Also such an architecture could be defined **after a conceptual analysis into a logical/functional architecture which is elaborated independently of any non functional and target constraints** (that is by supposing an ideal environment).
- An architecture that supports, e.g. structuring principles by grouping code implementing logically related properties, *especially by defining components as implementation of abstract data types*, allows the **definition of interfaces that are invariant between some parts**, say applicative ones, and some others, say the target support environment. This latter may itself be implemented as a set of implementations of abstract data types, whose properties define a "virtual ideal machine", and **define formally the separation of the applicative part from the underlying target system**.
- **independence** with respect to environment and/or target systems allowing definition of portable **components, or not dependent from target characteristics.** One sees that such property **can be gained "by construction"** if the architectural principles outlined above have been satisfied.
- The availability of a number of basic components is a condition for the building of larger scope components by using composition mechanisms according to the same architectural principles. It is even possible in the future to come to a definition of components with high abstraction and reuse value and still having these three fundamental properties.

2.11 HOOD AS A COMMON DEVELOPMENT FRAMEWORK

A commonly agreed observation on software development projects is that a unique method cannot cover all development activities. This is why HOOD has always to be put in work together with different methods either in the upper or lower phases of the design activities. Some projects came even to the conclusion that since the design activities are central in the development lifecycle, HOOD could be used as the framework for integrating the different development activities and methods.
Software development consists in fact in producing successive models of the future system using methods and models at different levels of abstraction that are each time better suited to the goals of a current development activity:

- **conceptual analysis models** are used in the upper phases and activities to describe the problem entities with terms, entities, abstractions that are from the domain, and in a way completely independent of any implementation choice.
- **design or architecture models** allow to describe the components of a solution that can be associated to a problem description in terms of structures, abstractions and behaviours.
- **implementation models** allow to describe the mapping of previous functional components onto an execution structure (that is tools and supporting services of the target system).

In order to produce all these models, a limited number of methods could be identified for a project, where HOOD could a central method to capture the reference representation of the system to be developed.

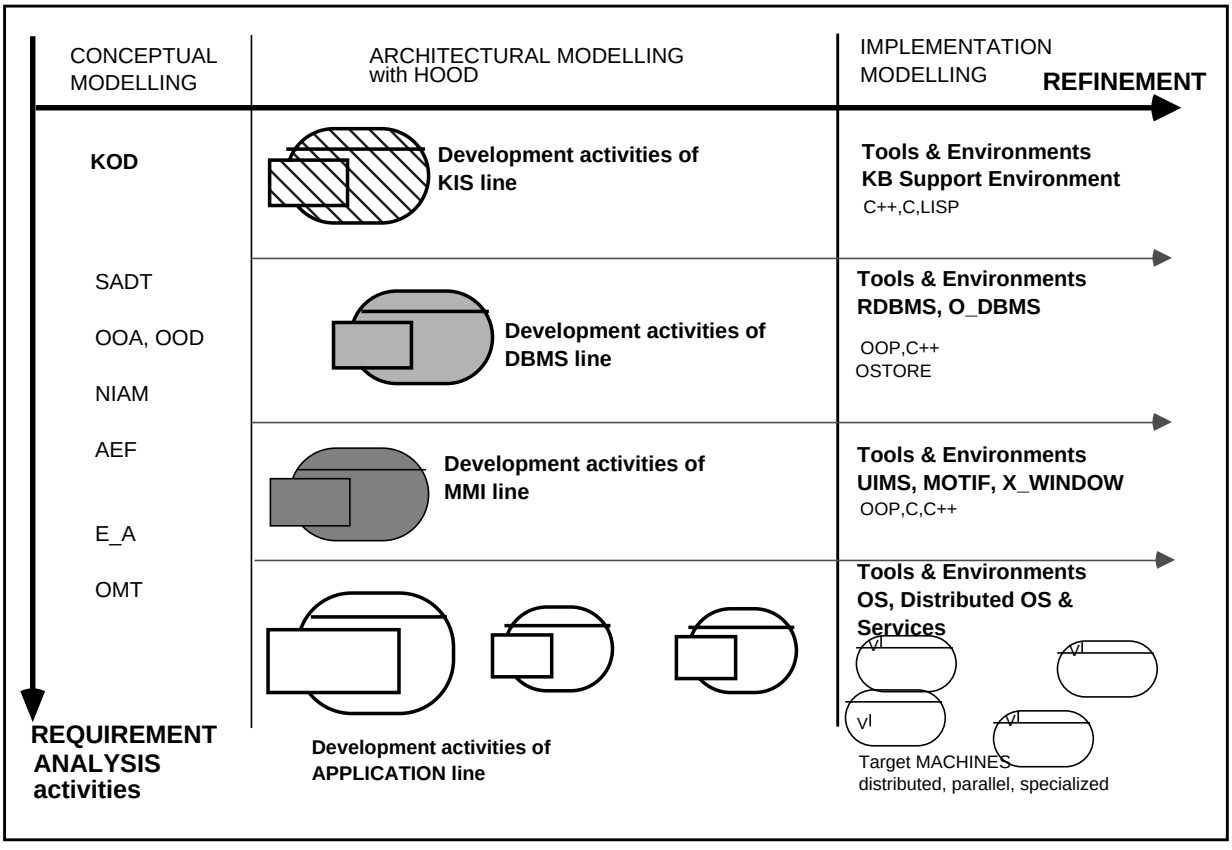


Figure 106 - Models and Components of a complex software architecture

Figure 106 - above summarizes the modelling activities of a development classified in three groups, each of which being supported by specific methods and tools:

- **Conceptual modelling activities** tent to grasp problem entities and try to give a formal model of them. In these activities one finds techniques and methods of requirement analysis and engineering.
- **Architectural modelling activities** tent to formalize associated solutions items. In these activities one finds design techniques and methods.
- **Implementation modelling activities** are refinement techniques leading to implementation on specific targets. In these activities one finds coding techniques (manual or automated through 4th generation tolls and languages)

All these different activities may be composed according to the understanding and the level of mastering of the development. Activities are however chained and triggered according to the following principles:

- COMPLEXITY and PROBLEM SOLVING REFINEMENT going from conceptual models towards implementation through design and architectural ones.
- TECHNOLOGICAL REFINEMENT along specialized lines of activities: the development of a data processing system may involve
  - **an MMI development line focusing on the development of the Man Machine Interface**, that uses the technology of UIMS (User Interface Management System or Generator) and windowing management standards.
  - **a DBMS development line focusing on the data handling and storing,** possibly tied to the knowledge that are manipulated. These activities are dealing with the definition of data models, Database schemes and rely on Relational and Object DBMS tools.
  - **an APPLICATION development line concerned with the development of the core application and interfaces** that corresponds to classical software engineering activities and which rely on host OS services and target system features.
  - **a KBS line** when the system comprises a Knowledge System and Inference system **associated to the development of KBS**

In *Figure 106* - ,the sequence of the different methods is not defined, but one can see that different methods shall be used for the conceptual modelling of the requirements (SA methods were intensively used, but now more and object oriented analysis methods jump in).
Using a standardized representation such as the HOOD one, allows the developer to have a pertinent representation of both the top-level decomposition of the system as well as for each of each component. This is an important progress towards the full mastering of the development process, because the actors of the development are now provided with a reference and communication framework, that is independent of the use of a specific technologies.
Moreover HOOD representations can be used as the communication and synchronisation tool for mastering the different development lines and ensuring proper final integration

2.11.1 INTEGRATION OF MULTIPLE-DEVELOPMENT TECHNOLOGIES

The modelling activity of an architecture is a main activity in the development of complex systems, and must allow a smooth transition from

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

Such developments may in fact include several lines of activities, each of which is relying on a different technical approach, but worked out and Adapted to the specific application domain. As an example, the MMI (Man Machine Interfaces) are developed more and more using an UIMS (User Interface Management System), and the data management systems, with the help of application generators (or 4th generation tools).
Hence a development of such a system is rather a set of more or less parallel lines of technology independent activities. These latter must be *scheduled, synchronised, and interface mastered* to support the associated management activities in a life cycle framework.
The interfaces between the different *subsystems* (or sometimes software layers) associated to the activity lines can be formalised as abstract data types that factor the exchanged data and provide for a common representation whatever the activity context. The use of HOOD, as the common representation formalism for the different lines jumps in naturally and brings several benefits:

- external representation (defining formally the interfaces of a component) standardised through HADT concepts,
- use (for the internals of the component) of the best suited formalism according to the technology supported by each activity line; whereas the relationship towards the other external technologies are always captured by HADT descriptions,
- definition of transformation rules from the common representation formalism towards an implementation formalism *allowing final code merging*. (for example ADT towards Object programming language, or ADT toward HOOD ODS and Ada, or ADT towards HOOD ODS and a sequential language.)

Figure 107 - below illustrates such a model represented with HOOD graphical formalism. Four objects associated to four development lines partition the system into technological modules that exchange data, that are again described through HADT objects represented as uncle objects where:

- the access interface to a datum (with all services provided to its clients) is formalised by the set of operations that manipulated the associated type (creation, update of a sub field, read, delete, checks, etc.) in a HADT object.
- further refinement of HADTS producing HADTS of less complexity up to terminal specification directly implementable in a target OO language, is based on the graphical extension and visibility restrictions outlined in section 4.3 above.
- *ADT_Applis_DATA* groups the definition of HADT objects associated to data exchanges between MMI and/or APPLICATION.
- *ADT_KNW_DATA* groups the definition of HADT objects associated to data exchanges between the inference system and the DBMS.
- *ADT_DBMS_DATA* groups the definition of HADT objects associated to data exchanges between MMI and/or APPLICATION with the DBMS, where:
  - either dataflows between DBMS and APPLICATION and/or MMI are direct instances of

abstract data types associated to the definition of the data model supported by the DBMS.

- either dataflows are build from abstract data types and define formally the interface between APPLICATION and/or MMI and the DBMS

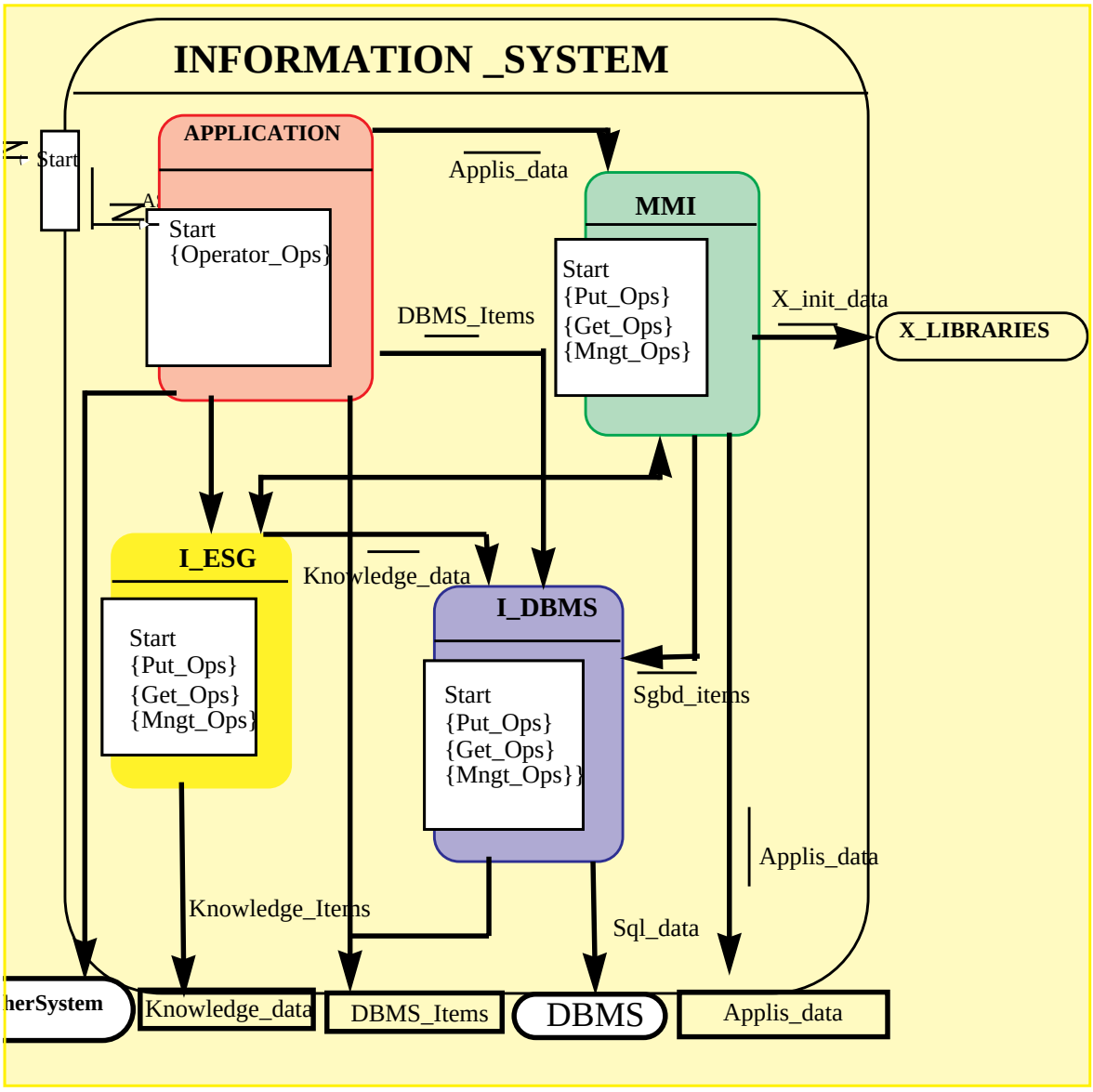


Figure 107 - HOOD Architectural Model of a complex information system

2.11.2 CONCEPTUAL AND BEHAVIOURS MODELLING

The features and properties of HOOD3.1 designs allow the implementation of a phased development approach relying on the following principles:

- **definition of an initial solution model** by abstracting away from implementation with well separated descriptions associated to the different characteristics of the model:
  - *the structural properties* are described in terms of HOOD objects and relationships within the scope of a system-configuration
  - *the functional properties* describing what a HOOD software does in terms of sequential executions are described in OPCSs
  - *the dynamical properties* describing the significant state transitions of a system within OBCS of objects.
- **Consistent refinement leaving initial models invariant**
  The invariance of an initial HOOD model is obtained through use and include relationship properties:
  - the include relationship allows *to refine a HOOD model by addition of new child objects, leaving their parent external specifications unchanged.*
  - the use relationship allows to refine the implementation of internals OBCS and OPCs and Types and Data by requiring services provided by Environment objects.

  The consistency between different model descriptions at different levels of details is ensured by:
  - the formalization of the ODS into:
    - external/public specifications and properties which are frozen as soon as the object is identified as a child within its parent.
    - internal properties, which can be refined without impacting on external specifications and either as in an implementation of a child service or in a target language implementation description
  - the ODS documentation concept which **integrates** the different characteristics of an object within description fields, where natural language or more formal notations can be used.The ODS into descriptions fields comprising:
    - informal textual description parts allowing expression of characteristics allowing the use of informal verification techniques.
    - more formal description parts (PSEUDO_CODE, CODE) allowing the use of specific notations to capture formally associated properties.
    - code description parts which directly reflect the translation of the properties into a target language.

The integration of object properties and supporting notations which is achieved in the ODS concept, together with the unique properties of the include relationship provide HOOD with powerful modelling and design features.

2.11.3 DYNAMIC BEHAVIOUR MODELLING

Dynamic behaviour modelling and evaluation is an important issue for real time and critical software, especially when one knows that most of errors detected on such systems come for bad mastering of dynamic behaviour.
The clear separation enforced by HOOD for the description of OBCS and OPCS structures makes it possible to:

- use suitable notation to capture, possibly formally, the behaviour associated to states and transitions defining the semantics of an object
- use classical sequential languages to describe the semantics associated to sequential processing within the OPCss.
  Note however that formal notations can also be used, namely algebraic specifications or mathematically based notations as illustrated using Z[GIOVA89]
- apply a development approach where the two lines of development are separated.

The separation of OBCS and OPCS parts can even be enforced down to the implementation into a target languages and systems. The code generation rules defined in the HOOD Reference Manual HRM[1]recommend to group the OBCS implementation associated to constrained operation into a dedicated target unit:

- in case of an Ada target this unit is named <object_Name>.OBCS and may implement both execution request and internal state constraints using the Ada tasking semantics.
- in case of non Ada targets (or when a non Ada executive is used) the unit <object_Name>.OBCS may implement the operation constraints through calls to a real time library or OS services providing the synchronization mechanisms needed (Semaphores, mailboxes, wait-for event or wake-up event services

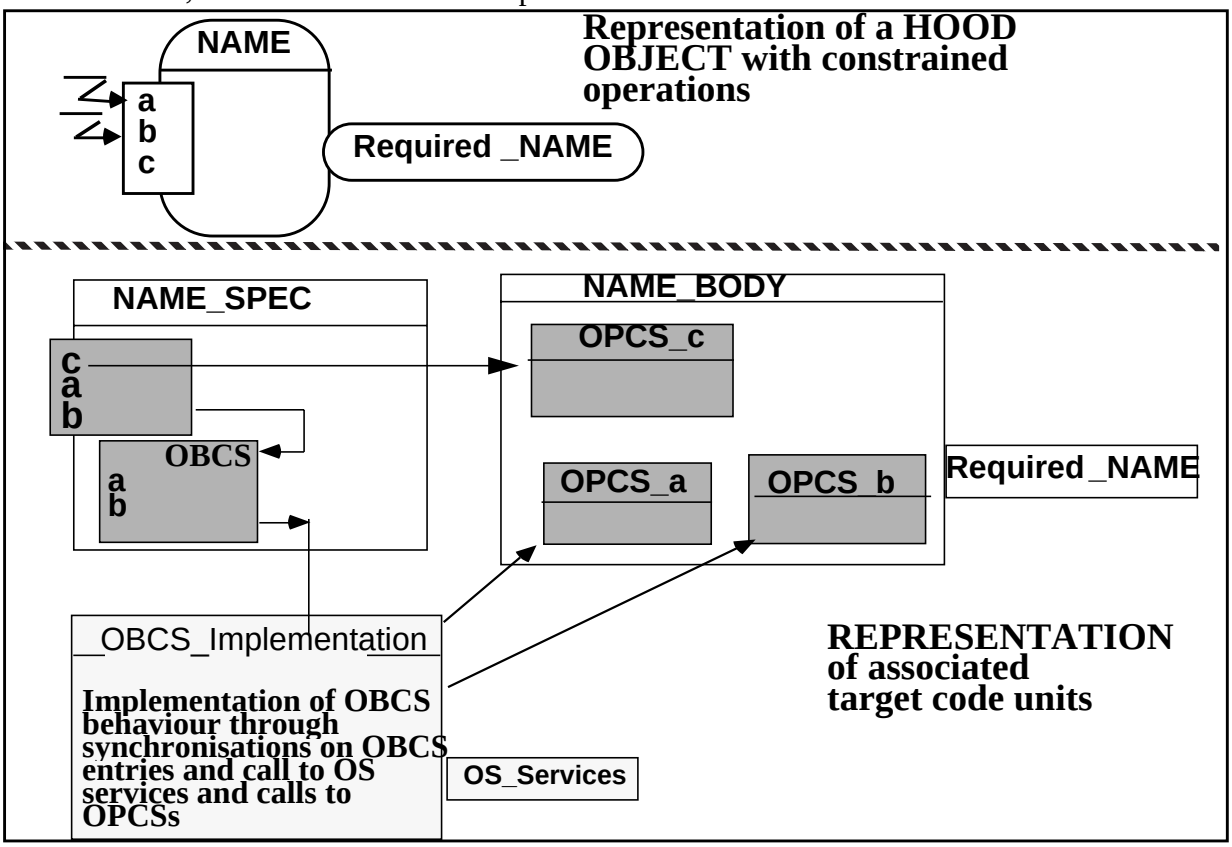


Figure 108 - Target Code architecture associated to OBCS and OPCSs

Figure 108 - gives a block diagram of such a target architecture. Target units may represent Ada package specification and body units. Note that constrained operations are mapped into OBCS entries and that non constrained operation are mapped directly into OPCS bodies.

2.11.3.1 Extended Object Execution Model

In the above section we gave the code generation principles for basic execution constraints attached to operations. However the implementation of more complex execution requests associated to communication protocols must also be defined. In *Figure 109 -* below an extended execution model in terms of HOOD objects, defines a standard architecture for implementation of any execution constraints, where:

- I_INTERFACE defines a set of interface units tasked with handling the execution requests (ASER,HSER,LSER,TOER) from clients, and to transmit these request to the OBCS as standardized events(OP_execution request event or OP-execution with time-out)
- OBCS is a target unit implementing a state transition model and which allows to control the triggering of OPCSs according to incoming requests and the internal state of the object as defined by the designer.(and using a suitable notation/formalism)
- O_INTERFACE is a target unit tasked with handling the execution of OPCSs on OBCS "enable" signals, and to acknowledge back when OPCS has completed.

The three units define possible process spaces and in case data (such parameters) exchanges are involved, we suggest to store and access it through an associated abstract data type implementation in a HOOD environment object. Such an implementation is either defined by the designer or may be generated automatically by HOOD toolsets.

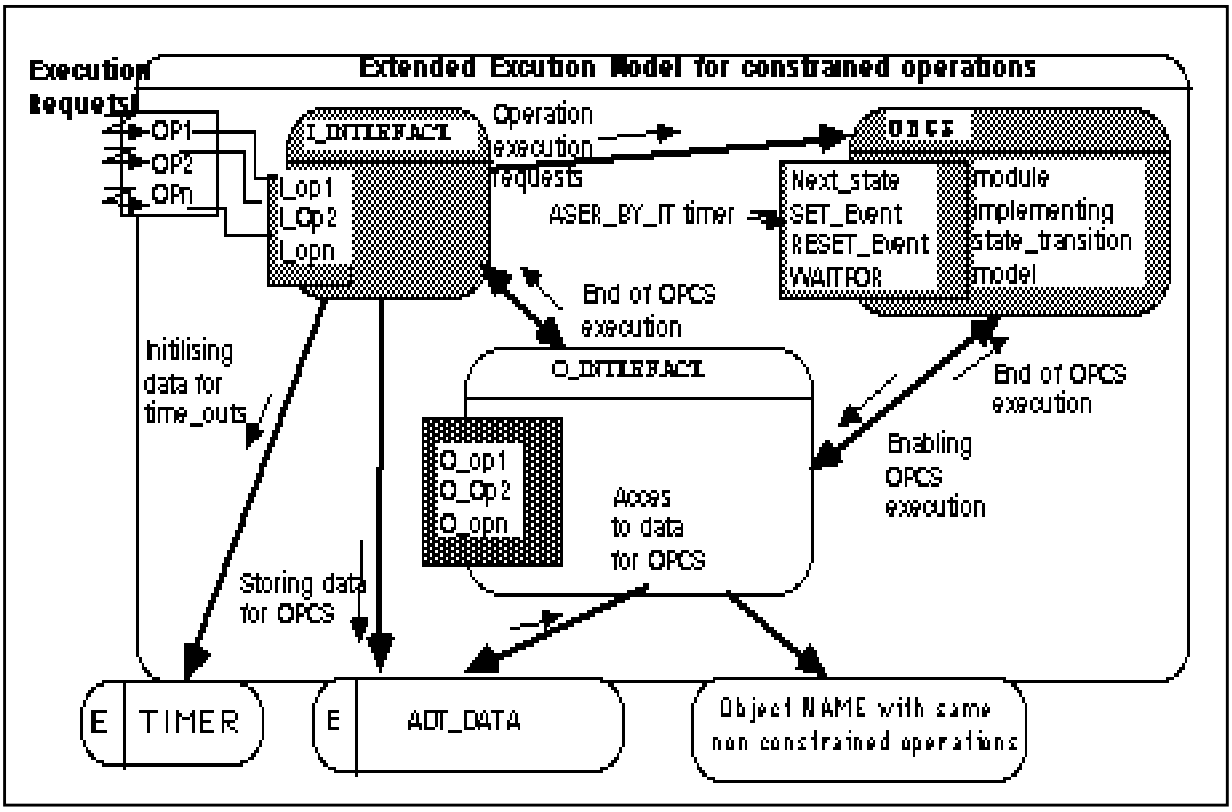


Figure 109 - Extended Execution model for constrained operations

Associated target models for the above units can be defined, in defining the OBCS as a synchronous implementation of state-transition system, associated to a set or I_INTERFACE and O_INTERFACE procedures:

- for an OBCS automata

```
loop
    Capture and store EVTS --events handling since last step
    Compute next STATE --
    Execute associated actions --(execute OPCS, emit new signals)
end loop;
```

Figure 110 - Automata Code modelling an Object OBCS

Such a synchronous approach is supported by notations (ESTEREL, LUSTRE, SIGNAL[BERRY88]:) and can be an implementation approach for other notations and techniques such as SDL and Petri nets

- for an I_INTERFACE procedure the implementation can be summarized as:

```
procedure I_OP_HSER is --call to a constrained operation
begin
    SEMAPHORE.P ; -- lock for exclusive access
        OBCS_AUTOMATA.Set_Event(Object_Name_OPname);
        wait_for(Object_Name_End_OPname); --wait until End of Execution
    SEMAPHORE.V; -- unlock
end I_OP;
```

Figure 111 - Principle of I_INTERFACE procedure Code

- for an O_INTERFACE procedure the implementation can be summarized as:

```
procedure O_OP_HSER is--associated to the operation requested
    begin
        OPCS_OP_HSER;  -- OPCS described  in "standard HOOD"
        OBCS_AUTOMATA.Set_Event(Object_Name_E_OPname);
    end O_OP_HSER;
```

Figure 112 - Principle of O_INTERFACE procedure Code

This model is only an illustration of implementation principles, but may easily be tuned according to particular constraints of the target:

- active waiting strategy (polling), sleep or wake-ups on events for HSER, LSER or TOER.execution requests
- step by step execution of the synchronous automata on event arrival or cyclic execution on timer interrupts at a frequency which is compatible with the one of event processing;
- execution of procedure O_OPi by the 'OBCS' process, or autonomous execution by active waiting processes, or execution triggered by event handling.

In the following we give the INTERFACE procedure for different protocol constraint operations

- HSER constraint: the client process must be blocked until end of operation execution, thus

```
procedure I_OP_HSER is
    begin
        AUTOMATA.Set_Event(Object_OP);
        wait_for(Object_End_OP) ; --suspends client until End of operation
end I_OP_HSER;
```

Figure 113 - I_INTERFACE Code for HSER constraint

and

```
procedure O_OP_HSER is
    begin
        OBJECT.OP;  -- OPCS defined by a standard unconstrained operation
        AUTOMATA.Set_Event(Object_End_OP);
    end O_OP_HSER;;
```

Figure 114 - O_INTERFACE Code for HSER constraint

- LSER constraint: the client process must be blocked until begin of operation execution, hence:

```
procedure I_OP_LSER is
    begin
        AUTOMATA.Set_Event(Object_OP);
        wait_for(Object_Begin_OP); --suspends client until Begin of operation
end I_OP_LSER;
```

Figure 115 - I_INTERFACE Code for LSER constraint

and

```
procedure O_OP_LSER  is
    begin --no Set_event (Begin_Op), =>AUTOMATA sends it back to waitfor
        OBJECT.OP;  -- OPCS defined by a same unconstrained operation
end O_OP_LSER;
```

Figure 116 - O_INTERFACE Code for LSER constraint

- ASER constraint: the client process must resume as soon as the requested operation has been performed, hence:

```
procedure I_OP_ASER is
    begin
        AUTOMATA.Set_Event(Object_OP);
end I_OP_ASER;
```

Figure 117 - I_INTERFACE Code for HSER constraint

and an associated O_OP_ASER procedure as for O_OP_LSER

- TOER_LSER constraint: the client process must resume on time-out overrun or on begin of

operation

```
procedure I_OP_LSER (time_out: in :=false) is
   begin
      TIMER.Set(Time_out_Value)); --sets a timer.
      AUTOMATA.Set_Event(LSER_TOER_OP);
      loop
         if AUTOMATA._Event(TO_OP) then -- time-out overrun event
            timed_out := true;
         exit;end if;
         if AUTOMATA._Event(Begin_OP_TOER_OP) then
            timed_out := false;
         exit;end if;
      end loop;
      TIMER.Reset;  -- resets timer.
end I_OP_LSER;
```

Figure 118 - I_INTERFACE Code for TOER_LSER constraint

and an associated O_OP_LSER procedure as for LSER without TOER

- TOER_HSER constraint: the client process must resume on time-out overrun or on completion of operation

```
procedure I_OP_HSER (time_out : in :=false) is
   begin
      TIMER.Set(Time_out_Value));--sets a timer.
      AUTOMATA.Set_Event(OP_HSER_TOER_OP);
      loop
         if AUTOMATA._Event(HSER_TOER_OP) then--time-out overrun event
            timed_out := true;
         exit;end if;
         if AUTOMATA._Event(End_OP_HSER_TOER_OP) then
            timed_out := false;
         exit;end if;
      end loop;
      TIMER.Reset;-- resets timer.
end I_OP_HSER;
```

Figure 119 - I_INTERFACE Code for TOER_HSER constraint

and an associated O_OP_HSER procedure as for HSER without TOER

2.11.3.2 Requirements for selecting control expression notations

In the above section we have outlined an approach where a primary condition for use of a notation to express control was its **ability to express State transitions**. Note that a designer needs only to define the behaviour of an object with respect to its internal functional semantics, using possibly formal notations; the burden of **handling execution requests processing can be supported through automatic code generation** as illustrated below.

Although use of formal notations would help capture and verify unambiguously the semantics, (namely by using associated environment and tools) it is also fundamental to be able to implement this **semantics straight down in the code**. We have found only few notations and languages supporting such features (ESTEREL, LUSTRE, SIGNAL), SDL, MCTL[HEI91]. However synchronous code generation for Predicate-transitions Petri nets could be easily defined (see *Figure 122 -*).

As more formal notations are used, we would like to rely directly on their formal basis to verify directly the properties of the models produced (without generating code and testing). This requires supporting tools for:

- **checking consistency of one given model**; the checks will rely on intrinsic properties defined by the semantic definition of the notation and/or language use to model an OBCS.
- **checking consistency of composition of models:** as a full system behaviour is the composition of elementary object behaviours, the ability of composing OBCS would be useful.
- **validation** of behavioural descriptions against models from requirements analysis or models produced at another level of description (e.g. equivalence of a model describing a HOOD parent object would be checked against the composition of behaviours of its child objects).
- Validation is currently relying on simulation of behavioural descriptions and pragmatic comparison of equivalence through test scenarios. A formalism based on a strong mathematical model of concurrence would allow to **perform formal calculi** and **proofs** on the described behaviours. Some notations such as synchronous languages (ESTEREL,SIGNAL) have now tools to support such checks based on automata behavioural equivalence (using the INRIA AUTO tool) and/or through temporal logic formula as described in [LECMP89]. Nevertheless such proof systems have exponential complexity for concurrent systems. A.Heibig[HEI91] states that his compositional semantics of control machine have linear complexity. Advances in the area of logic checkers and concurrent system evaluation tools will strongly impact on the choice of formal notations for developing critical systems.

2.11.3.3 Defining an Associated Verification Process

Once the approach has been experimented, it can be automated and supported by a selected set of toolsets. The verification process is, anyway, improved, and better quality and confidence in the resulting software will be achieved.

- OBCS behavioural descriptions can be:
  - produced, **verified through simulation** if notations used to express OBCS behaviours are supported by simulation environments.
  - **verified through execution** when code is generated. Test cases and scenarios elaborated in previous development phases may be reused.
  - developed and tested separately from the development of the sequential software. As a result of this separation, **more verification and testing effort** (in terms both of time spent and skills of developers) **can be allocated** to this critical part of the software.
- Integration will be based on three strategies:
  - **several synchronous automata working in parallel** (seeFigure 120 - below).In this case the development is conducted as usually down to the identification of terminal objects The associated OBCSs are expressed in the chosen notation and the associated code contains as many automata as terminal OBCSs. The final tuning must be done automata by automata, during integration activities.
  - **one unique synchronous automata** defined as the “code” composition of all the elementary ones.This is obtained by **merging all states and events of elementary automata into one new set,** thus reducing the minimal number of processes from N*3 to 3. The new automaton is not the product of the others but only the union allowing better mastering and tuning of the units I_INTERFACE, O_INTERFACE and AUTOMATA.(a centralized

strategy particularly suited to embedded cyclic systems)

- **one unique synchronous automata** defined as the **formal composition** of all the others. This can only be done with very specific notations and under given hypothesis. For e.g ESTEREL is able to compute a compositional automata from several others; however one must check that the 'synchronism hypothesis' is still valid. Also there is a lot of trouble to expect due to combinatory explosions of states when doing the cartesian product of the elementary automata. MCTL (Modular Control and Temporal Logic)[HEI91] is a promising technique but with poor tool support and practically no “real case study” feedback. Petri net based techniques are also promising, since composition of Petri nets is well mastered.[GIOVA89][PALU90][VIEL89]

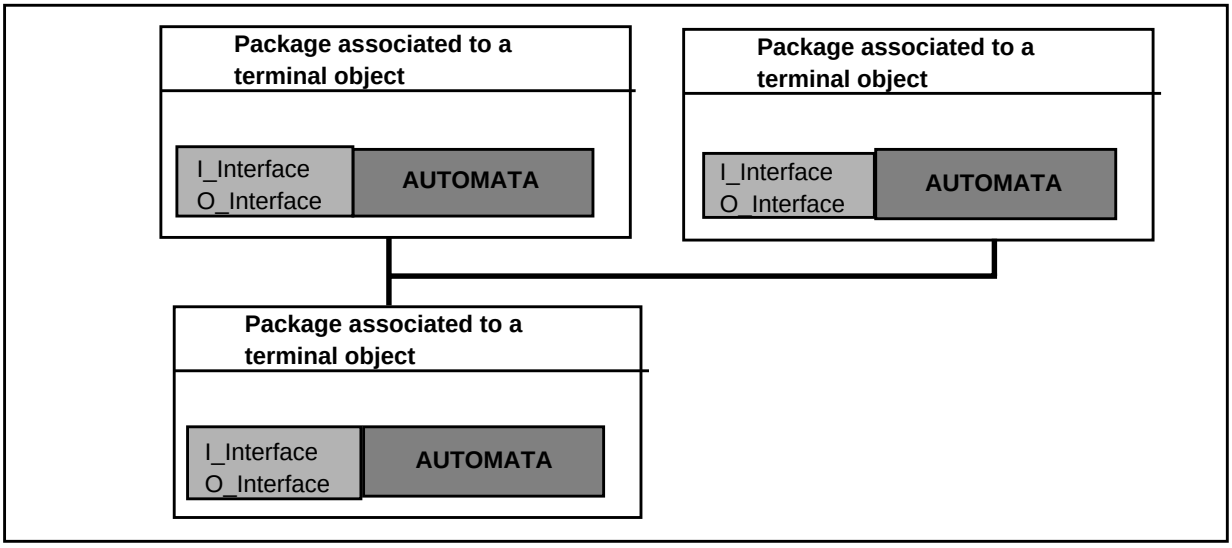


Figure 120 - Integration based on several concurrent AUTOMATA

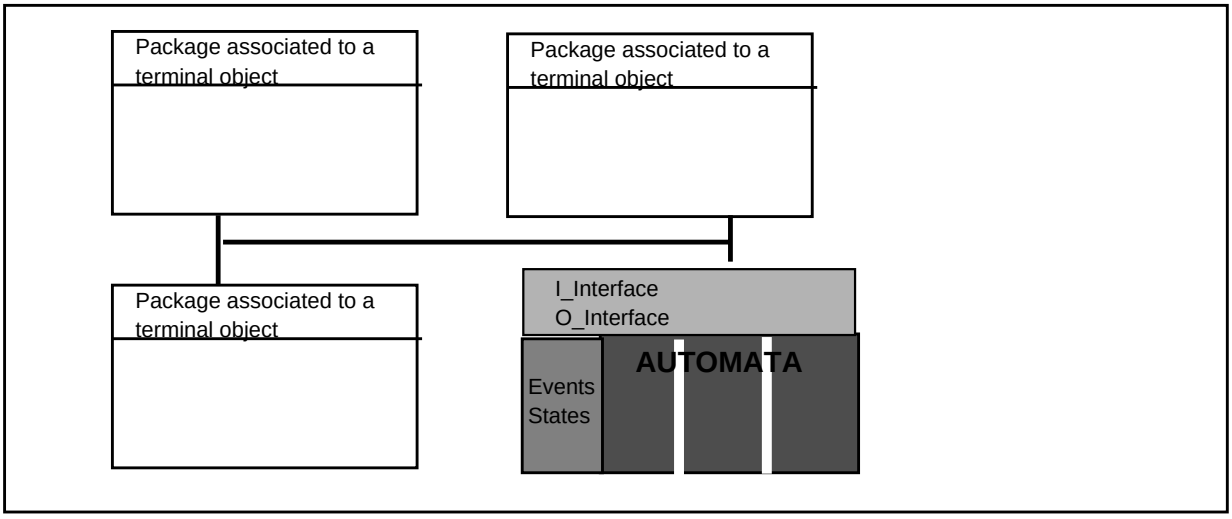


Figure 121 - Integration based on merge of AUTOMATA

The advantage of such approaches is that the full system behaviour could be modelled, expressed and verified formally, with HOOD bringing structure and decomposition support, and formal notations a sound mathematical support.

2.11.3.4 Synchronous Automata Code for Predicate Transition nets

Begin_ OP and End_OP variables are associated to places in order to store the number of tokens. SET-EVENT procedures increment Begin_OP or End_OP variables as token are created.

The OBCS automata is implemented as one execution flow which examines all transitions of the network, and eventually fires some of them. Interpreting is reduced to evaluation of the conditions and execution of the operations associated to transitions.

For each operation appearing on the transitions of the net, a procedure EXECUTE(B_OP) will be defined in the AUTOMATA package associated to the object. (see Figure 122 -below). Also for each condition associated to a transition, in the net, a function returning a boolean must be associated.(see function Cond inFigure 122 - below)

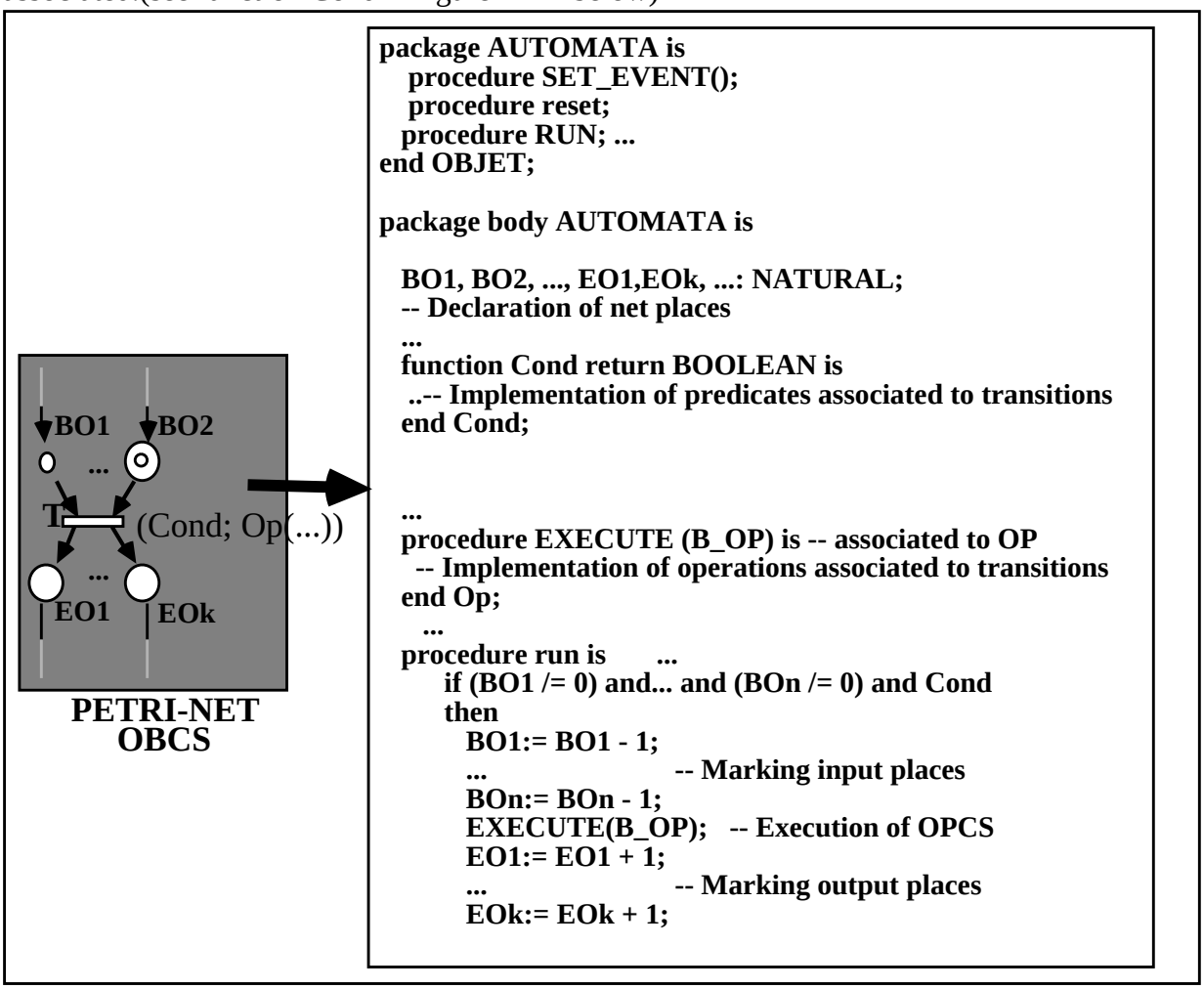


Figure 122 - Ada Code generation for PTN

2.11.4 PERFORMANCE EVALUATION

2.11.4.1 Annotating a HOOD design for "automatic" performance analysis

Our purpose is to define a method for inserting performance data into a HOOD design in order to generate a performance model which can be used to check that the proposed design meets the performance requirements. The description of this method is divided into 3 parts:

- description of the different kinds of performance related data which must be inserted into a HOOD design,
- description of the way performance data can be inserted into a HOOD design,
- description of the techniques used to annotate with the different kinds of performance data.

2.11.4.2 Kinds of performance related data

The Figure 123 - describes entities used for performance specification, design modelling (including hardware aspects) and performance evaluation together with relationships between these entities.

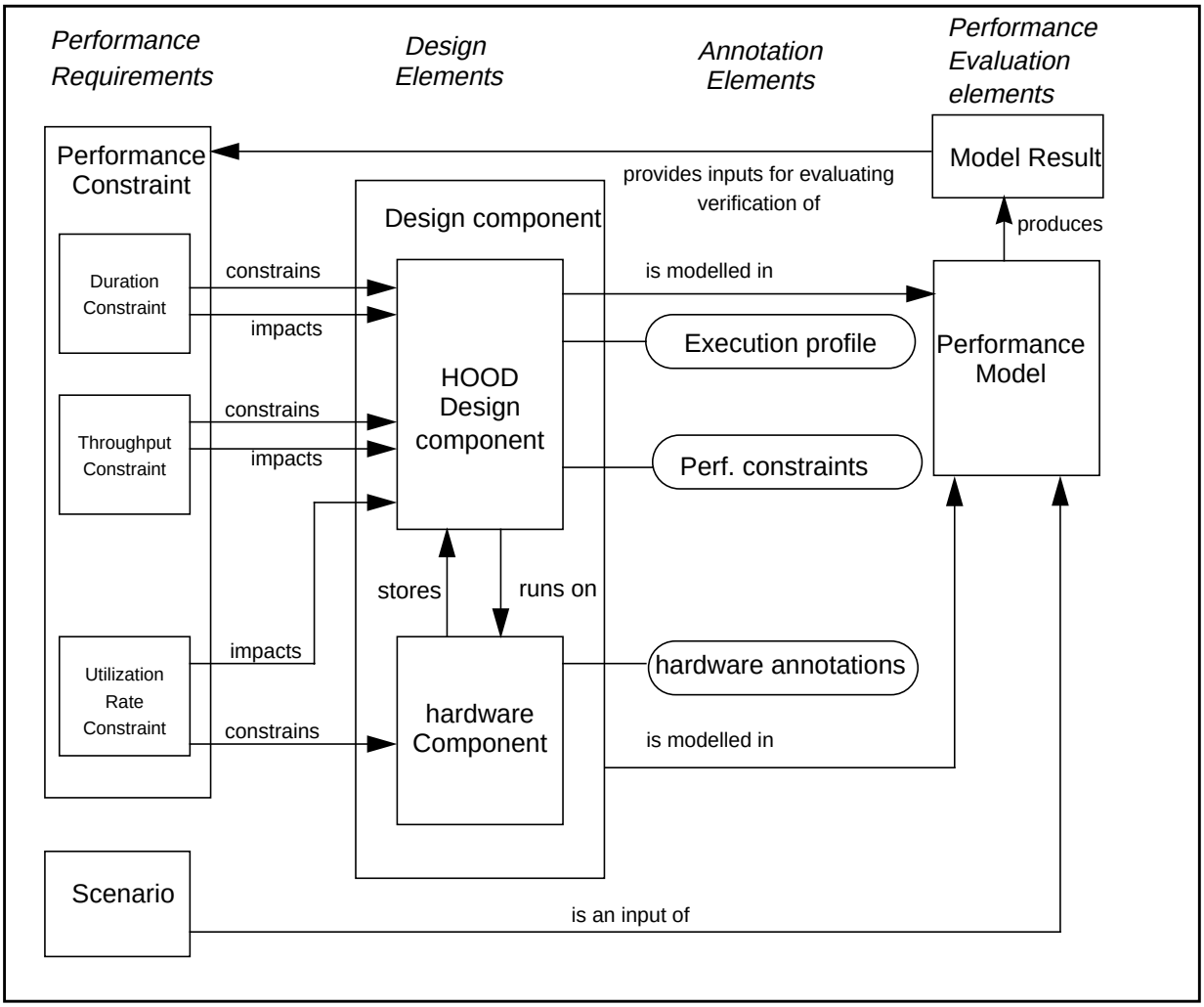


Figure 123 - : from performance requirements and design elements to performance model

The *Figure 123 -* shows that two different kinds of data have to be inserted into HOOD design components:

- It is interesting, for traceability reasons, to annotate HOOD design components with the **performance constraints** which either "constrain" or "impact" them. The relationships "constrain" and "impact" are two different kinds of traceability links between performance constraints and design components. To explain the difference between them we can say that "constrain" identifies design components which are explicitly constrained (these components implement the function which is identified in the performance constraint) and "impact" identifies components which may interact with the constrained ones.
- **Execution profiles** can be attached to HOOD design components for describing the way software accesses hardware resources.

2.11.4.3 Inserting annotations into a HOOD design.

The simplest way of associating annotations with HOOD design components is to insert them in HOOD object descriptions via structured comments.
A brief description of the Object Description Skeleton is provided below. We have emphasized in this ODS description what parts of the ODS need to be identified before the insertion of annotations.

2.11.4.4 Performance Annotations for a HOOD design

2.11.4.4.a Annotating a HOOD design with performance requirements.

- Annotation by duration and throughput constraints
  For each HOOD operation **constrained** by a constraint, we propose to associate the constraint with the operation in the field IMPLEMENTATION_CONSTRAINTS of the corresponding object description (see Figure 124 -)

.

IMPLEMENTATION_OR_SYNCHRONIZATION_CONSTRAINTS
. . .
-- $ **PERFORMANCE CONSTRAINTS**
-- One description of constraints for each operation impacted by one or several constraints
-- $**OPERATION** Name_op1
-- One constraint description for each constraint impacting the operation
-- $**DURATION_CONSTRAINT | THROUGHPUT CONSTRAINT** constraint_expression[1]
. . .
. . .
-- $ **END_PERFORMANCE CONSTRAINTS**
. . .

1. informal expression or semi-formal expression based on the use of some attributes defined in the taxonomy of performance requirements

Figure 124 - : Annotation with performance constraints

- Annotation by utilization rate constraints
  As these constraints **constrain** hardware components, it is not possible to associate them with HOOD design components since the method only addresses software components; these annotations shall be associated with the description of the physical architecture which identifies hardware components and connections between them.

2.11.4.4.b Annotating a HOOD design with execution profiles.

For each HOOD operation **impacted** by performance constraints, we propose to annotate the DESCRIPTION field of the corresponding OPERATION CONTROL STRUCTURE with the operation execution profile in the way described on Figure 125 -.

```
OPERATION Name <| parameter part>
DESCRIPTION
-- $EXECUTION PROFILE
-- Structured annotations providing data needed to build performance models
. . .
```

Figure 125 - : Annotation with an execution profile.

Three kinds of statements may be used to describe an execution profile:

- **Resource oriented statements:** these statements describe resources consumption. Three statements corresponding to three kinds of resources have been defined:
  - $WCET statement: processing resource consumption in terms of worst case execution time
  - $IO statement: I/O resource consumption
  - $CALL[1] statement: Communication resource consumption

- **Control oriented statements**: these statements describe control structures including sequences of resource-oriented statements. Three kinds of statements corresponding to the more classical control structures have been defined:
  - $LOOP statement to build a loop in the execution profile.
  - $BRANCH to build a selection statement (similar to the “switch” statement of the C language).
  - $WAIT statement to wait for a timer expiration.

- **Measurement oriented statements:** these statements are used to put measurement points in an execution profile:
  - $DURATION_MEASUREMENT_POINT to measure the execution duration of a software component.
  - $THROUGHPUT_MEASUREMENT_POINT to measure the throughput at a particular point in an execution profile.

An example of such annotations is provided in Figure 130 -

[1] We use call statements to describe communication resource consumption because data are always exchanged through use relations between operations.

2.11.4.5 Building a performance model from an annotated HOOD design.

The purpose of this chapter is to describe how an annotated HOOD design could be used to build a performance model. The main steps of the derivation process are shown on Figure 127 -.

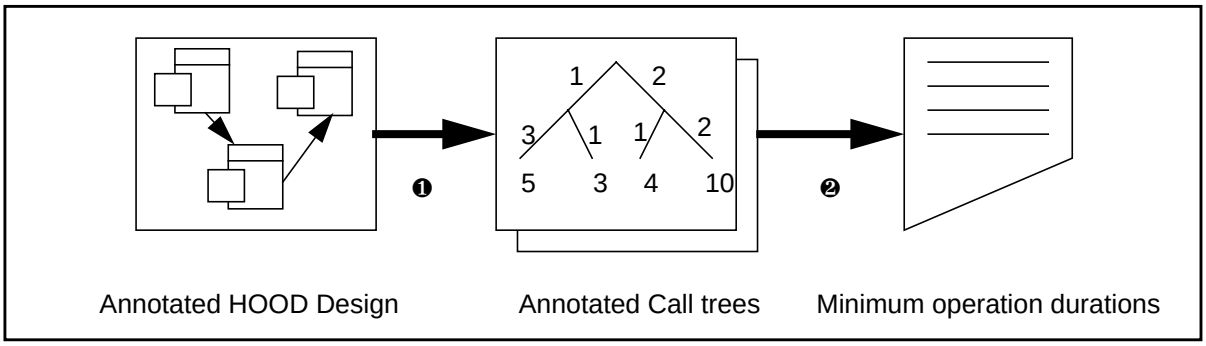


Figure 126 - Estimating worst case execution time

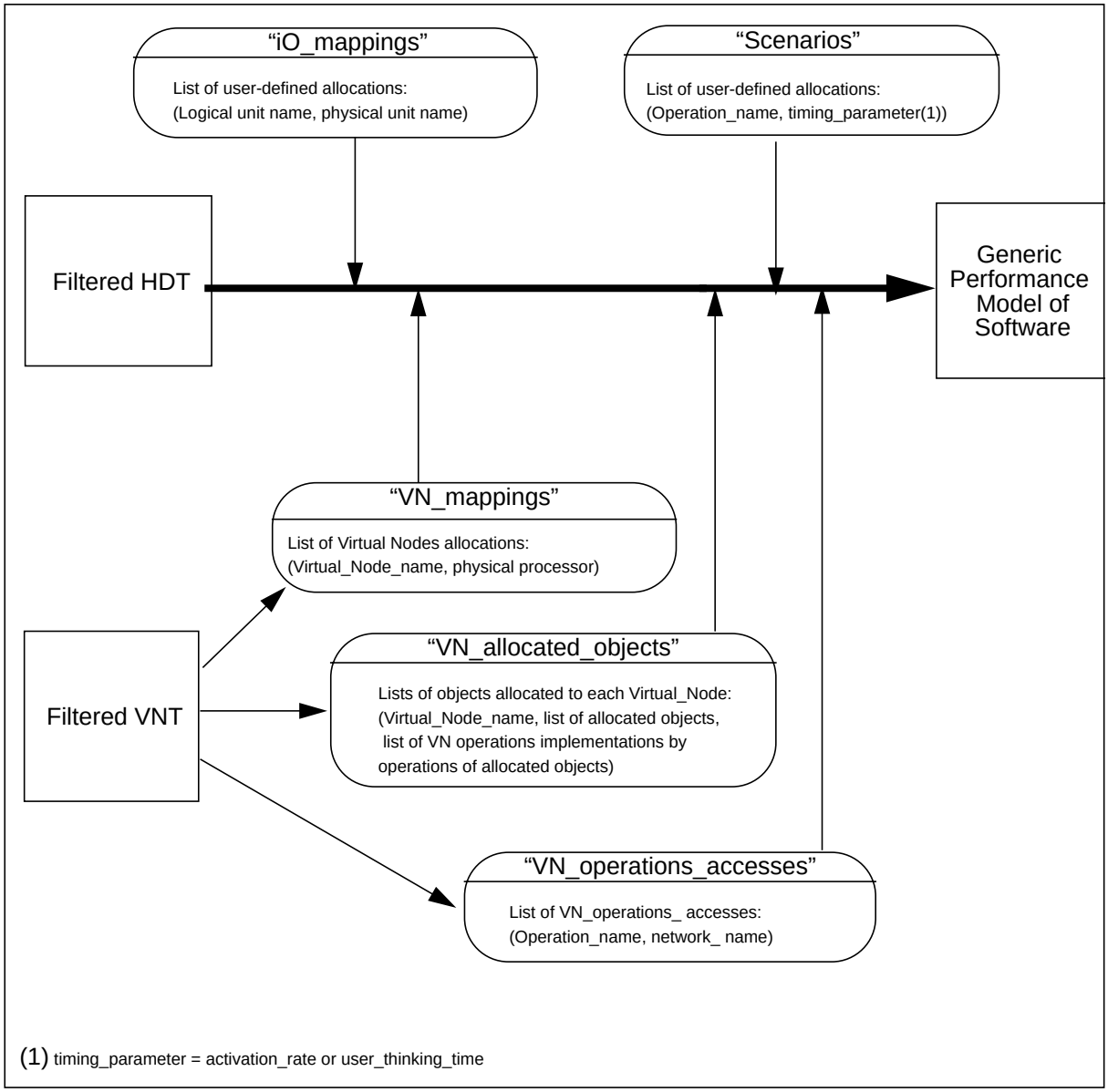


Figure 127 - : Generation of a performance model from an annotated HOOD design

2.11.5 TIMING ESTIMATION AND SCHEDULABILITY ANALYSIS

A timing estimation approach may be used to verify the schedulability of an application and to perform CPU time budget analysis via structured annotations within the HOOD design.

This approach is based on the HRT theory defined in[HRT]. The HRT theory implies an identification of all threads and the definition of their execution constraints in terms of priority, deadlines,... From these information, it is possible to analyse the schedulability of the application and / or to simulate it.

The idea is to annotate HOOD objects with timing estimations called worst case execution time (WCET), schedulability informations (e.g. priority, deadlines,...) and to specify operation flowgraphs based on nominal and selected critical operational scenarios. WCETs may latter on be consolidated by real estimations based on the code inserted within the OPCSs. In such a case special features of a compiler may used.
The operation flowgraphs defined in each objects are consolidated and reduced (i.e. via a maximisation of the different branches) depending of the overall operation calling tree.

These information may used both to generate CPU budgets by extraction of relevant information from the ODSs and to perform schedulabity analysis by extraction and computation of execution skeletons to be provided to a Schedulability analyser.

Figure 128 - and Figure 129 - describe the process model of the building of the threads execution skeletons with estimations of worst case execution times to be used for schedulability analysis while Figure 130 - provides an example of an ODS with such annotations.

.

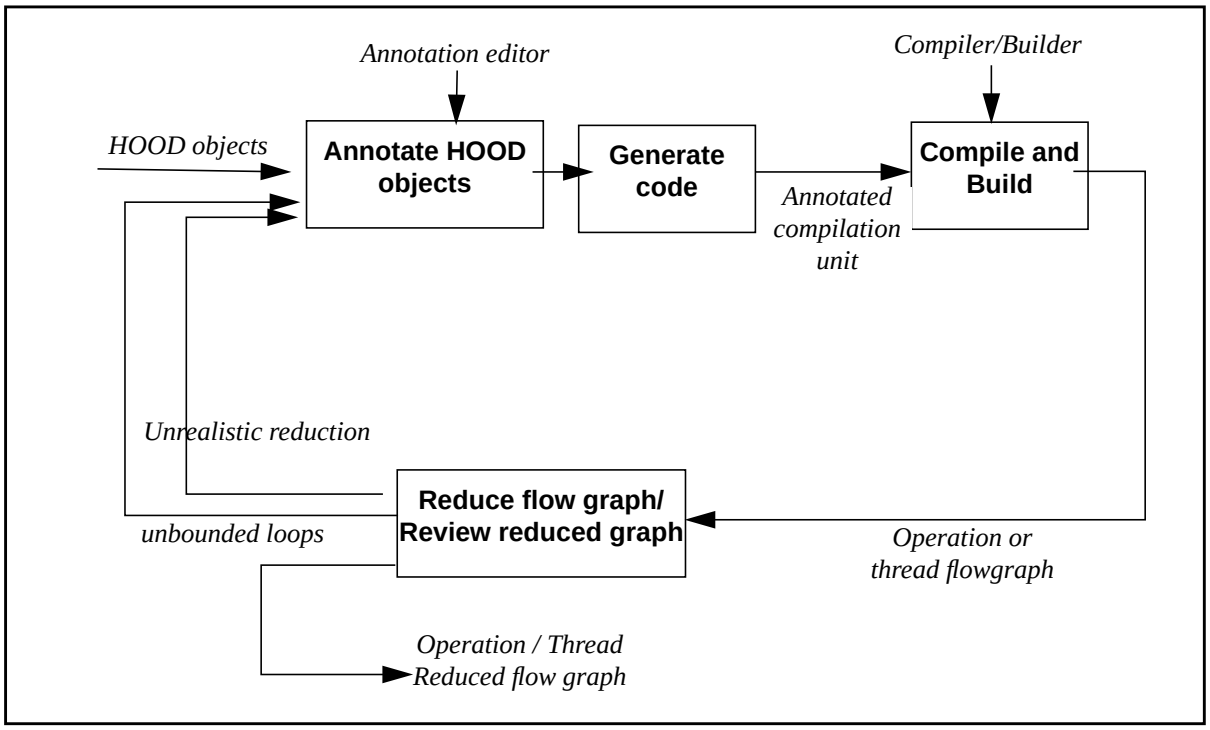


Figure 128 - Timing estimations: Analysis of each HOOD object

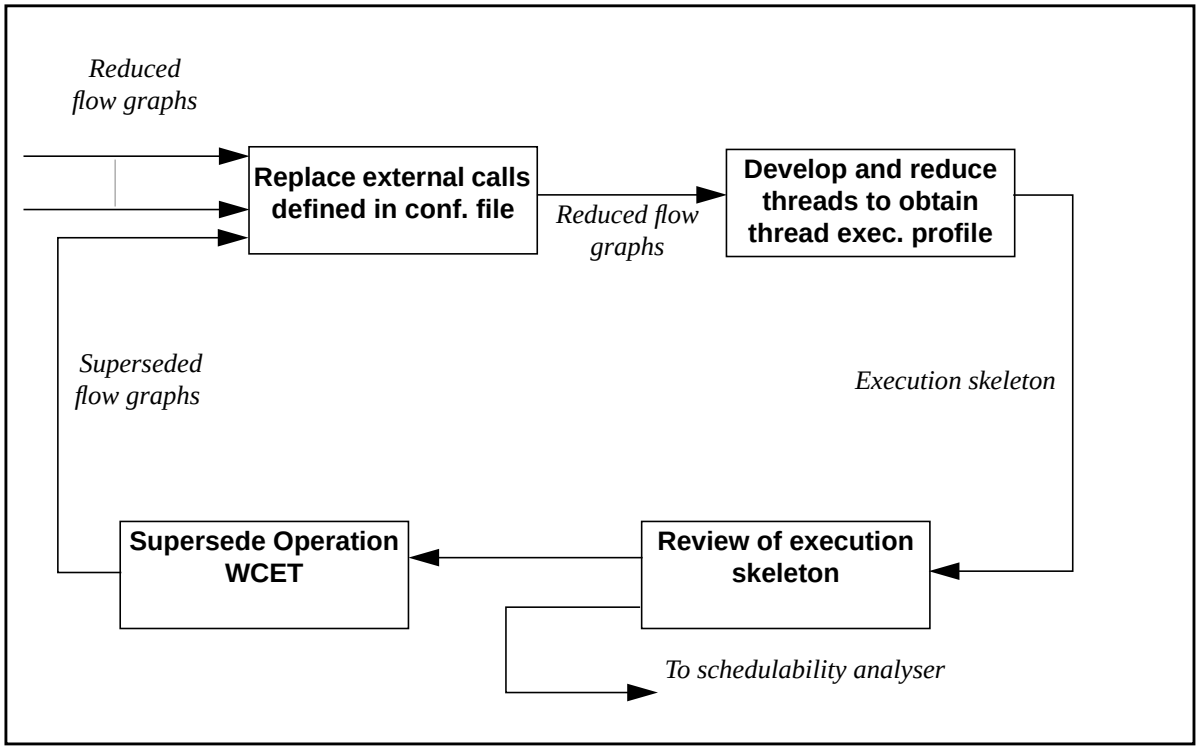


Figure 129 - Timing estimations: Build overall execution skeleton

```
OBJECT Objet1 IS ACTIVE
DESCRIPTION
-- ....
IMPLEMENTATION_OR_SYNCHRONIZATION_CONSTRAINTS
-.....
- - Annotation with performance constraints.
-- $DURATION_CONSTRAINT DC1 “The response time of Operation1 shall be less than 5 ms”
.....
OPERATION Operation1
DESCRIPTION
    -- -- This field contains the description of the operation.
    -- -- It may also contain an execution profile such as the following one:
    -- $WCET(10 ms) [estimated or computed worst case execution time]
    -- $IO(Read,File1,10 KB)
    -- $LOOP(10)
        -- $WCET(1 ms)
        -- $CALL(Operation2, SEND 1 KB, GET 2 KB)
    -- $END_LOOP
 USED_OPERATIONS
                    ......
                    ...
END_OPERATION Operation1
```

Figure 130 - Example of HOOD annotations for performance and timing estimations

2.12 TARGET LANGUAGES

In *Appendix 1.1.6* of present document the general principles for code generation are given and the current section will not present them again. However we shall bring here complementary information and illustrations.

2.12.1 HOOD TO TARGETS IMPLEMENTATION PRINCIPLES

HOOD ODSs are formatted texts, grouping object properties, including a number of target dependent fields (directly entered as target code). HOOD code generation principles are then to generate:

- a software architecture in terms of target modules/units:
  - INCLUDE relationships are implemented by target encapsulation mechanisms if they exist, or by simple file inclusion
  - USE relationships are implemented through target mechanims to control visibility intra and inter units. For some targets an include strategy with EXTERN and REF clauses may have to be defined.
- target code skeletons to implement associated visibility relationships from HOOD required interface descriptions
- ODS code fields inserted in the associated code skeletons

Another principle is the one of preserving the identifiers used in the HOOD ODS descriptions within the target code. This may raise some difficulties especially when the target is not Ada. In order to limit these difficulties it is recommended to follow "project defined" coding standards in documenting target dependent fields.

2.12.2 HOOD TO ADA CODE GENERATION

Since HOOD was initially developed for Ada targets, “doing Ada” within HOOD should not be too troublesome. Difficulties however have been experienced:

- the implementation of provided TYPES of a parent object since there is no means for renaming Ada types. The HOOD/Ada[HADA] study recommends to use as far as possible a translation scheme through derived types, or subtyping with automatic generation of “lost declarations” when the former scheme is not possible.
- the standard schemes for active object support uses heavily tasking what is often felt as non efficient, or even not possible when dynamic behaviour is to be supported by non Ada or RT OS. The general principle is then to have a clear separation between the sequential code of the OPCS and all additional code needed to implement state and protocol constraints. This has major impacts on the way a system can be developed and tested since sequential development can be conducted “in parallel” with the development of the dynamic architecture and supporting code. In the following we shall illustrate application of such solutions both for full Ada targets and for targets using “no Ada tasking”
- Finally due to the variety of targets and project constraints, a number of code generation pragmas are often used to drive the code generation away from standard schemes. These pragmas may be useful but care shall be taken to avoid destructuring of the code architecture

making it difficult if not impossible to maintain them from HOOD ODS or reverse back into HOOD descriptions.

2.12.3 SEQUENTIAL LANGUAGES (C, C++, FORTRAN AND ASSEMBLERS)

2.12.3.1 Operation Signatures

A first convention has been adopted by all toolset builders in using the Ada syntax to describe operations signatures whatever the target language; this greatly helped in the SIF definition (Standard InterChange Format).
In order to deal with syntactic specificities such as the C pointer marker '*' or the C++ reference marker '&' we recommend however to use additional associated HOOD type definition and naming conventions.

In order to stress operation attached to an OOP class, an additional parameter of type the class may be added in the operation signature(the name of this additional parameter is *me* in HOOD4)

Thanks to Ada syntax and semantics a HOOD code generator has then all information for generating highly reliable and readable code associated to operation signatures.

2.12.3.2 Exceptions

When the target language does not support exceptions we recommend first to limit the use of exceptions in those HOOD designs and to use a specific EXCEPTIONS module (of a *HOOD RUN TIME LIBRARY*) to mimic the Ada exceptions handling.
An implementation of a "*HOOD RUNTIME*" module has been defined for C and C++ targets by some projects, and have now been included in HOOD4 definition . One of such solution requires a renaming (by means of C macros) of all operations able to propagate exceptions to their clients.

- *Implementation of EXCEPTIONS.*
  For each HOOD exception declaration, an enumerated value is defined of type T_X_VALUE which is provided by an environment object called *EXCEPTIONS.*

  - Raising an exception[1] is by calling *EXCEPTIONS.RAISE* (X_EXCEPTION_NAME), an operation that updates the exception current (global) value.
  - Testing an exception is done by a call to *EXCEPTIONS.SET* with a possible branching to an exception handler by *GOTO exception.*
  - In C these constraints may be transparent through use of predefined macros like the following:

```
≠define procedure() /* redefine a procedure name raising exceptions*/
    EXCEPTIONS.Reset( ); /* set current exceptions to OK*/
    X_Procedure(); /* effective call renamed by prefixing it with "X_"
    if EXCEPTIONS.SET( ) goto exception; /* return test for branching to the exception handler
≠endef /*( if omitted an error will be detected by the C compilerr*/
```

Figure 131 - Sample code of testing exceptions in client code

[1] In C++, raise is replaced by a *throw* statement and testing by a *catch* statements.

2.12.3.3 Generation rules for passive objects

Passive object implementation presents no difficulties, and the encapsulation mechanisms of the languages will be extensively used (FORTRAN libraries, PASCAL module, C programs and modules, assembler compilation units)

2.12.3.4 Generation rules for active objects

HOOD3.1 defines support of constrained operation by means of an OBCS unit (tone or several tasks) that handle control and communication protocol between client and server threads.
When target with no ADA tasking are used, protocol constraint operation can only be implemented by using a HOOD RUN TIME library that implements inter-process communications associated to the definition of HOOD protocol constraints.
Such implementation rules and illustrations can be found in *Appendix A3* of present document and have been standardised through use of *HOOD runtime environments objects* such as OBCS, FSM, and EXCEPTIONS, and proposed for inclusion in the HOOD4 definition.

2.12.3.5 Entity Naming

Since a number of target languages restrain the identifier size and do not support dotted notation problems may come up about the naming of identifiers. The solution is a strict naming policy for HOOD Designs targeting language other than Ada, e.g. Fortran[2].

2.12.3.6 Conventions for C targets

C references to HOOD entities by dotted notation can be replaced by an appropriate “global name”.One convention could be to replace each prefix of a dotted chain by “ObjectName__” the double ‘__’ avoiding conflicts with a C identifier including already one ‘_’.
The code architecture into C modules enforces the following rules:

- provided and exported resources to client modules, especially operation signatures are generated in .h files
- Associated implementation are specified in .c files.
- visibility relationships are implemented through “INCLUDE” clause that recall required resources from associated .h files.

The only constraint for the HOOD designer will be to make explicit references to required operations in following these conventions. (at OPCS code level only)

2.12.3.7 Code generation for Environments objects

The following code generation rules shall apply for environment objects:

- for a code generation of a HDT no environment code should be generated
- Code generation for a full system, shall consist in generating code for all HDT or CDT of a system configuration. No code associated to Environment object outside of the System con-

[2] FORTRAN 90 is said to have made enormous progress in this fields.

figuration shall be generated.

- HOOD objects, modelling environmental software, shall have naming rules compatible with the code identification or interface of the real code as provided by third parties.
  For example all references to standard packages provided by the Ada development environments must enforce the naming associated to identification of provided services. Naming of TEXT-IO is standard, but not the attributes of SYSTEM.
- When modelling OS interfaces, the HOOD naming scheme shall be the one of the existing interface library (e.g. SYSTEM_CALLS).

2.12.4 OBJECT ORIENTED LANGUAGE TARGETS

OOPL languages can be introduced smoothly into current HOOD3.1 definition. HOOD can integrate OOP designs and implementations with minor extensions. In the following we first present how to use OOP with current HOOD definition, and then the extension that has been proposed to the HTG for HOOD4 definition.

2.12.4.1 Designing for OOPL implementation (HOOD3.1)

HOOD may first been used as a software partitioning tool defining module interfaces. These modules may define **class interfaces** and their implementation and may be directly coded using target OOPL (e.g C++, or Eiffel). These classes are seen as TYPES by other HOOD clients objects, which declare DATA INSTANCES of that classes/types in the DATA field of their ODS. OPCSs are code including call to methods that apply on that HOOD DATA.

2.12.4.2 Other Conventions for C++

Naming conventions for HADT/C++ classes: we recommend to adopt a naming schema for classes and types in order to distinguish between classes and types, e.g. the following naming schema has been found useful in several projects:

- all class names are prefixed by "T"
- all type names are prefixed by "T_"
- all exception names are prefixed by "X_"

Exceptions support is similar as the one suggested for C.

2.12.4.3 Implementation of state constrained operations

An OOP class implementing sate constrained operations could be coded with an additional attribute FSM as an instance of the class TFSM, handling a finite state machine for each instance of that class. The construction of that class should initialize the machine states according to data describing the allowed state transitions. See *Appendix A3.2.1* for a full illustration in Ada, and note that it would be just a matter of rewriting that code in an OOP language.

2.12.4.4 Implementation of protocol constrained operations

See *Appendix A3.2.2* for a full illustration in Ada of the implementation principles of protocol constrained operation. Inter process communication primitives would also have to be provided by means of a RUNTIME library for a given target system/host OS. Such a run time library is already provided by some HOOD toolset vendors targeting C, and is now specified in the HOOD4 definition.

2.13 FULL ADA CODE ILLUSTRATION

2.13.1 TERMINAL PASSIVE OBJECT

Let us consider a simple terminal passive object that implements a STACK as an abstract data, encapsulating the data that represent the stack and providing operations to manipulate them.

2.13.1.1 ODS Definition for passive STACK object

A passive STACK object, implementing an Abstract Data for a STACK, may be represented as in *Figure 132* - , with two operations PUSH and POP

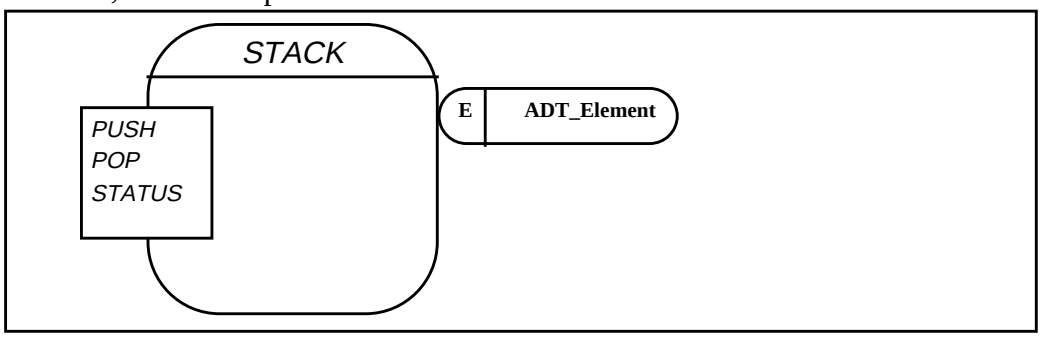


Figure 132 - Graphical representation for the object Stack

The ODS of *Figure 133* - shows the ODS declarations of provided and required types and operations and exceptions

.

```
OBJECT STACK IS PASSIVE -- l
DESCRIPTION --Implementation of a abstract Data STACK with no encapsulation of associated data
IMPLEMENTATION_CONSTRAINT S -none
PROVIDED_INTERFACE
   OPERATIONS
        Push (Element : In T_element);
        Pop(Element : Out T_element);
        Status return T_Status;
   EXCEPTIONS
        X_EMPTY raised by POP;
        X_FULL raised by PUSH;
---------------------------------------------------- END OF STACKS USER MANUAL ----------------------------
REQUIRED_INTERFACE
   OBJECT ADT_ELEMENT
   TYPES
   T_Element;--STACK requires type T_Element, provided by ADT_ELEMENT.
```

```
INTERNALS -- hidden part of the object:
   OBJECTS None;  --NO CHILD OBJECTS
   TYPES
   type T_Status is (BUSY, IDLE, UNDEFINED); -- definition of provided type to hold the status of the stack code.
   type T_DATA is array(integer <>) of ADT_Element.T_Element; -- --data structure to hold values of the type
   DATA
        TOP  : integer:=1; -
        SIZE: integer:=100;
        DATA: T_DATA (1 .. SIZE);
        STATUS: T_Status;
   CONSTANTS None; -- no internal constants
   OPERATIONS  None; -- no internal operations
OPERATION CONTROL  STRUCTURES
OPERATION  Status return T_Status  is  --OPCS of POP
   DESCRIPTION
   Get status of class STACK code
   USED_OPERATIONS None;-
   PROPAGATED_EXCEPTIONS  -- None;-
        CODE-- in Ada
        begin
             return STATUS;
   END  --of opcs STATUS

OPERATION  POP  is  --OPCS of POP
   DESCRIPTION
   Remove an Element from the data structure STACK_DATA
   USED_OPERATIONSNone;--
   PROPAGATED_EXCEPTIONS
        X_STACK_UNDEFINED;
   PSEUDO_CODE--
        if  [the stack is full ] then
             raise X_FULL;
        else
             [push the element on the stack]
        end if;
   CODE-- Ada
        begin
        if TOP>0 then
             STATUS:=busy;  -- STACK code is BUSY
             Element :=STACK_DATA(TOP);
             TOP:=TOP-1;
             STATUS:=idle;-- STACK code is IDLE
        else
             STATUS:=undefined;  --STACK code  UNDEFINED
             raise X_EMPTY; --impossible case
        end if;
   END  --of opcs POP
OPERATION  PUSH is  --OPCS of operation PUSH
   DESCRIPTION
   Put  an  élément onto STACK
   USED_OPERATIONS  None;--
   PROPAGATED_EXCEPTIONS
        X_STACK_UNDEFINED;;--
   PSEUDO_CODE--none see the code
   CODE -- (Ada)
        begin
        if  TOP<STACK.SIZE then
             STATUS:=busy;  --class code in BUSY state;
             TOP:=TOP+1;
             STACK_DATA(TOP):=Element
             STATUS:=idle;--class code in IDLE state
        else
             STATUS:=undefined;  -class code in UNDEFINEDstate
             raise X_FULL; --impossible case
        end if;
   END  -- de l'OPCS de PUSH
END_OBJECT STACKS
```

Figure 133 - ODS of passive STACK object

2.13.1.2 Ada code generation for passive STACK object

2.13.1.2.a Ada Specification Unit

```
--REQUIRED OBJECTS
with ADT_ELEMENT; -- visibility on type T_Element
package STACK is
   procedure Status  return T_Status;
   procedure  PUSH (Element : in T_Element);
   procedure  POP(Element : out T_Element);
   X_FULL : exception;  --raised by PUSH
   X_EMPTY : exception ; -- raised by POP
end STACK;
```

Figure 134 - Ada specification Unit for passive object STACK[1]

We can note that the provided operations are translated into procedure or function specification keeping the parameters and the return type unchanged; and provided exception declarations are translated in exception declarations.

2.13.1.2.b Ada Body Unit

```
with TEXT_IO;--USED OBJECTS
package body STACKS is
--internal Types
   type T_Status is (BUSY, IDLE, UNDEFINED);
   type T_DATA is array(integer <>) of ADT_Element.T_Element;
--internal DATA
        SIZE: integer:=100;
        DATA: T_DATA (1 .. SIZE);
        TOP: integer:=1; -
        STATUS: T_Status ;

   procedure PUSH( Element : in T_Element)is
   begin
   TOP := STACK'LAST;  -- use of  Ada Attribute functions
   SIZE_STACK:=STACK'RANGE;
        if  TOP<SIZE_STACK then
             STACK.STATUS:=busy;  -- STACKis used by some one ;
             TOP:=TOP+1;
             STACK.DATA(TOP):=Element ; -- put Element
             STACK.STATUS:=idle;-- STACK code is idle;
        else
             STACK.STATUS:=undefined;  --STACK code is in undefined state
             raise X_FULL;
        end if;
   end PUSH;

   procedure  POP (Element : out T_Element)is --similar to that of PUSH seen above

   procedure  Status  return T_Status is
   begin
        returnSTATUS;
   end STATUS ;
end STACK;
```

Figure 135 - Ada body Unit for passive object STACK

In this package body, we can see that the following translation rules have been applied :

- Internal types and constants are mixed together in the order of which they appear in the ODS, allowing correct Ada references from each other.
- internal data are translated without any changes.

1.

- the OPCS of an operation is translated into a procedure of function body, as such. Note however that the user must supply the keyword begin[2], but not the keyword end in the ODS code section.

To see some more about translation rules, and especially about private typês, lets now modify the previous example so as :

- to design an object that implements a STACK abstract data type, and exporting the ots type T_STACK
- using an external type T_Status defined in the environement object PROJECT_ENV

the associated ODS would be the following.

2.13.1.3 ODS Definition for passive STACKS

A passive STACKS object, implementing an Abstract Data Type for STACK data, may be represented as in *Figure 136* - , with two operations PUSH and POP. We have added also an environement object PROJECT_ENV for providing the definition of the type T_Status.

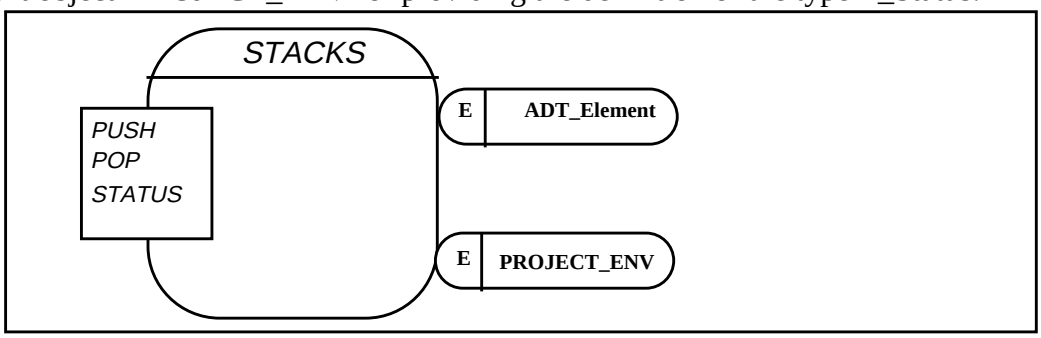


Figure 136 - Graphical representation for the active object Stacks

The ODS of *Figure 137* - shows the ODS declarations of provided and required types and operations and exceptions.

```
OBJECT STACKS IS PASSIVE -- l
DESCRIPTION  --Implementation of a abstract Data Type STACK with no encapsulation of data instances
IMPLEMENTATION_CONSTRAINTS  none
PROVIDED_INTERFACE
   TYPES
        type T_STACK(MAX: integer) is  private ;                --definition of type T_STACK as private
        OPERATIONS
        Push (Stack : In Out T_stack; Element : In T_element);
        Pop(Stack : In Out T_stack; Element : Out T_element);
        Status(STACK : in out T_STACK) return T_Status;
   EXCEPTIONS
        X_EMPTY raised by POP;
        X_FULL raised by PUSH;;
---------------------------------------------------------- END OF STACKS USER MANUAL -----------------------------
REQUIRED_INTERFACE
   OBJECT ADT_ELEMENT
   TYPES
        T_Element;--lSTACK requires type T_Element, provided by ADT_ELEMENT.
   OBJECT PROJECT_ENV
   TYPES
        T_States;--STACK requires type T_STATES, provided byt PROJECT_ENV
```

[2]. because a procedure may declare local data, in which case a generator may not know it and may misplace the begin statement.

```
INTERNALS -- hidden part of the object:
   OBJECTS None;  --NO CHILD OBJECTS
   TYPES  --private and  internal types
   type T_DATA is array(integer <>) of ADT_Element.T_Element; -- --data structure to hold values of the type
   type T_STACK(MAX: integer) is                    --definition of private  type T_STACK
   record
        DATA: T_DATA (1 .. MAX);                    -- only needed space is declared
        TOP: integer:=1; -
        SIZE: integer:=MAX;
        STATUS: PROJECT_ENV.T_Status;
   end record; -- a client declares stack objects as Client_Stack: T_STACK(150);

   CONSTANTS None; -- no internal constants
   DATA None; because this is an abstract type implementation with no data.
   OPERATIONS  None; -- no internal operations

OPERATION CONTROL  STRUCTURES
OPERATION  Status return T_Status  is  --OPCS of POP
   DESCRIPTION
   Get status of class STACK code
   USED_OPERATIONS None;-
   PROPAGATED_EXCEPTIONS  -- None;-
        CODE-- in Ada
        begin
             return STATUS;
   END  --of opcs STATUS
OPERATION  POP  is  --OPCS of POP
   DESCRIPTION
   Remove an Element from the data structure STACK_DATA
   USED_OPERATIONSNone;--
   PROPAGATED_EXCEPTIONS
        X_STACK_UNDEFINED;
   PSEUDO_CODE--
        none
   CODE-- Ada
        begin
        if STACK.TOP>0 then
             STATUS:=busy;  -- STACK code is BUSY
             Element :=STACK_DATA(STACK.TOP);
             STACK.TOP:=STACK.TOP-1;
             STATUS:=idle;-- STACK code is IDLE
        else
             STATUS:=undefined;  --STACK code  UNDEFINED
             raise X_EMPTY; --impossible case
        end if;
   END  --of opcs POP
OPERATION  PUSH is  --OPCS of operation PUSH
   DESCRIPTION
   Put  an  élément onto STACK
   USED_OPERATIONS  None;--
   PROPAGATED_EXCEPTIONS
        X_STACK_UNDEFINED;;--
   PSEUDO_CODE--
        none
   CODE -- t(Ada)
        begin
        if  STACK.TOP<STACK.SIZE then
             STATUS:=busy;  --class code in BUSY state;
             STACK.TOP:=STACK.TOP+1;
             STACK_DATA(STACK.TOP):=Element
             STATUS:=idle;--class code in IDLE state
        else
             STATUS:=undefined;  -class code in UNDEFINEDstate
             raise X_FULL; --impossible case
        end if;
   END  -- de l'OPCS de PUSH
END_OBJECT STACKS
```

Figure 137 - ODS of passive STACK S.

In the above ODS , the type T_STACK is provided as a private type, whose full declaration is only given in the INTERNALS.The associated package can be the following:

2.13.1.4 Ada code generation for passive object STACKS

2.13.1.4.a Ada Specification Unit

```
--REQUIRED OBJECTS
with ADT_ELEMENT; -- visibility on type T_Element
with PROJECT_ENV; -- visibility on type T_Status
package STACKs is
   procedure Status (STACK : in T_STACK) return T_Status;
   procedure  PUSH (STACK : out T_STACK; Element : in T_Element);
   procedure  POP(STACK : in T_STACK;Element : out T_Element);
   X_FULL : exception;  --raised by PUSH
   X_EMPTY : exception ; -- raised by POP
private
   type T_DATA is array(integer <>) of ADT_Element.T_Element;
   type T_STACK(MAX: integer) is
   record
         DATA: T_DATA (1 .. MAX);
         TOP: integer:=1; -
         SIZE: integer:=MAX;
         STATUS: PROJECT_ENV.T_Status;
   end record;
end STACKS;
```

Figure 138 - Ada specification Unit for passive STACKS[3]

2.13.1.4.b Ada Body Unit

```
with TEXT_IO;--USED OBJECTS
package body STACKS is
   procedure PUSH(STACK : in out T_STACK; Element : in T_Element)is
   begin
         TOP := STACK'LAST;  -- use of  Ada Attribute functions
         if  TOP<STACK'RANGE then
               STACK.STATUS:=busy;  -- STACKis used by some one ;
               TOP:=TOP+1;
               STACK.DATA(TOP):=Element ; -- put Element
               STACK.STATUS:=idle;-- STACK code is idle;
         else
               STACK.STATUS:=undefined;  --STACK code is in undefined state
               raise X_FULL;
         end if;
   end PUSH;

   procedure  POP (STACK : in out T_STACK;Element : out T_Element)is
-- code as entered in the code section of the OPCS-------------------------------------------
   --similar to that of PUSH seen above
   end OPCS_POP;

   procedure  Status (STACK : in T_STACK) return T_Status is
   begin
         return STACK.STATUS;
   end STATUS ;
end STACKS;
```

Figure 139 - Ada body Unit for passive STACKS

The translation rules for the generation of the package body have not changed, except that the declaration associatde to the private part of the type have not to appear in the package body.

3.

So far we have seen how to translate provided interface, required interface and internals, let us now introduce constraints on the operations PUSH and POP.

2.13.2 TERMINAL ACTIVE OBJECT

2.13.2.1 ODS Definition for active object STACKS

An active STACKS object, implementing an Abstract Data Type for STACK data, may be represented as in *Figure 140 -* , with operations PUSH and POP *HSER* and *STATE* constrained , thus enforcing :

- HSER : delegation of execution of the operations by a server thread, the client resuming its execution only when PUSH or POP are completed by the server.
- operation PUSH and POP activation conditions according to the behaviour specified in *Figure 141 -*

.

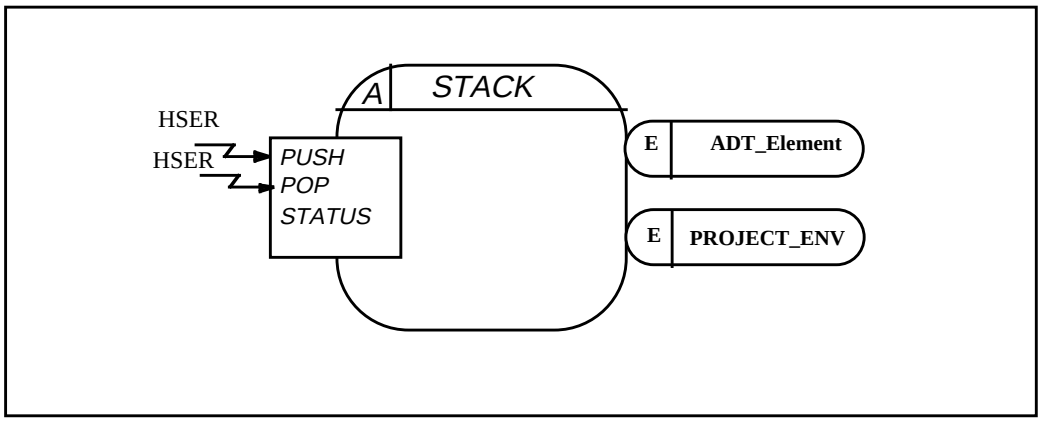


Figure 140 - Graphical representation for the active object Stack

The STACKS object behaviour can be described by a state transition diagram as in *Figure 141 -* , where transitions are exclusively associated to operation execution.

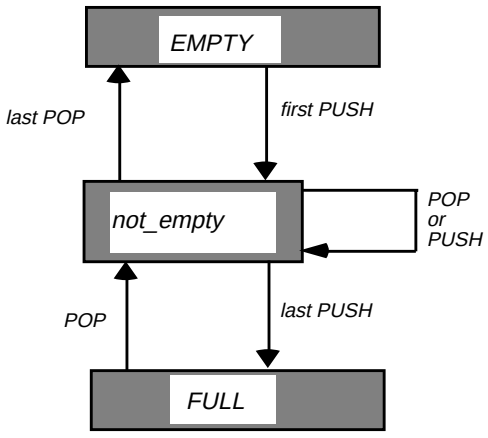


Figure 141 - Object behaviour for STACK

The OBCS can be reduced in this case into one Ada task implementing both the HSER protocol between the client thread and the server OBCS thread by means of rendez-vous entries, and the state transition diagram of *Figure 142 -* by means of guards computed from the state of the object.

The associated Ada code could be:

```
task body OBCS is
begin
loop -- code OBCS Ada de STACK
    select -- choice according to current state of the Stack given as parameter
    when not (IS_EMPTY(Stack : In  T_Stack) ) =>
    accept POP(Stack : In Out T_Stack; An_Element : out T_Element) do-
            OPCS_POP(Stack ,An_Element);            --associated OPCS body
    end POP;
    or when not (IS_ FULL(Stack : In  T_Stack) ) =>
       accept PUSH(Stack : In Out T_Stack; An_Element : in T_Element)  do
                OPCS_PUSH(Stack, An_Element);    -- associated OPCS body
       end PUSH;  -- release caller process
    end select
end loop;
end OBCS;
```

Figure 142 - Ada OBCS code for stack

In case no STATE constraints were associated to the operations, one would simply suppress the **guards** in the **accept** statements of the OBCS code.
In case we would only handle state constrained operations (with no HSER protocol) then :

- HOOD3 proposes a standard translation rules as with HSER constraint
- this was unacceptable by users, because of tasking overhead. That is why we have proposed a solution "with no ADA tasking" using a simple finite state machine as detailed and illustrated in section 2.14.1 and in *Appendix A3.1.2*.
  This approach has been adopted by the HTG for the HOOD4 definition which now clearly separates the state constraints[4] from the protocol constraints.

The ODS below shows the declarations associated to the OBCS, and declarations of provided and required types, where :

- the T_STACK is nomore declared private, thus illsutrating the T_STACK definition as ordinary, with visible structure type.
- we also introduced two internal operations IS_FULL and IS_EMPTY checking the variable TOP of the T_STACK structure, and used for computing the state of the object and the associated the guards in the accept statements of the OBCS code parts.

[4] HOOD3 had never distinguished between pure state constraint operations and protocol ones. This ambiguity is now resolved in HOOD4.

```
OBJECT STACKS IS ACTIVE -- l
DESCRIPTION --Implementation of an abstract data type STACK with no encapsulation of data instances
IMPLEMENTATION_CONSTRAINTS-- use of full Ada
PROVIDED_INTERFACE
   TYPES
   type T_DATA is array(integer <>) of ADT_Element.T_Element; -- --data structure to hold values of the type
   type T_STACK(MAX: integer) is                          --definition of type T_STACK
   record
        DATA: T_DATA (1.. MAX);                           -- only needed space is declared
        TOP: integer:=0; -
        SIZE: integer:=MAX;
        STATUS: PROJECT_ENV.T_Status;
        STATE:
        end record; -- a client declares stack objects as Client_Stack: T_STACK(150);
   OPERATIONS
        Push (Stack : In Out T_stack; Element : In T_element);
        Pop(Stack : In Out T_stack; Element : Out T_element);
        Status(STACK : in out T_STACK) return T_Status;
   EXCEPTIONS
        X_EMPTY raised by POP;
        X_FULL raised by PUSH;
OBJECT CONTROL  STRUCTURE -- visible part of OBCS
   DESCRIPTION  PUSH and POP operations should only be activated when the state of STACK is non(EMPTY) for POP or
non(FULL) for PUSH
   CONSTRAINED_OPERATIONS
   PUSH constrained by HSER;
   POP constrained by HSER-
---------------------------------------------------------- END OF STACKS USER MANUAL -----------------------------
REQUIRED_INTERFACE
   OBJECT ADT_ELEMENT
   TYPES
        T_Element;--STACKS requires type T_Element, provided by ADT_ELEMENT.
   OBJECT PROJECT_ENV
   TYPES
        T_Status;--STACK requires type T_Status, provided byt PROJECT_ENV
```

Figure 143 - ODS Illustration of STACKS

The INTERNALS part of the STACKS ODS in *Figure 144* - shows the declaration of internal types, data and OBCS code.

```
INTERNALS -- hidden part of the object:
   OBJECTS None;  --NO CHILD OBJECTS, no internal types, constants or data
   OPERATIONS
        IS_FULL(STACK : in  T_STACK) return boolean;
        IS_EMPTYL(STACK : in T_STACK) return boolean;
   OBCS
        PSEUDO_CODE
        loop  -- ADA translateion of the state transition diagram in Figure 141 -
             select
                  when not (IS_FULL) =>accept POP;
             or   when not (IS_EMPTY) =>accept PUSH
             end select
        end loop
        CODE  --code of OBCS
        task body OBCS is
        begin
        loop -- code OBCS Ada de STACK
             select -- choice according to current state of the Stack given as parameter
                  when not (IS_EMPTY(Stack : In  T_Stack) ) =>
                  accept POP(Stack : In Out T_Stack; An_Element : out T_Element) do-
                       OPCS_POP(Stack ,An_Element);              --associated OPCS body
                  end POP;
                  or when not (IS_ FULL(Stack : In  T_Stack) ) =>
                  accept PUSH(Stack : In Out T_Stack; An_Element : in T_Element)  do
                            OPCS_PUSH(Stack, An_Element);              -- associated OPCS body
                  end PUSH;  -- release caller process
             end select
        end loop;
        end OBCS;
```

```
OPERATION CONTROL  STRUCTURES

OPERATION IS_FULL return boolean  is
   DESCRIPTION check current object state
   USED_OPERATIONS None;-
   PROPAGATED_EXCEPTIONS  -- None;-
        CODE-- in Ada
        begin
             if STACK.TOP=STACK.SIZE then return true; else return false end if;
   END  --of opcs IS_FULL

OPERATION IS_EMPTY return boolean  is
   DESCRIPTION check current object state
   USED_OPERATIONS None;-
   PROPAGATED_EXCEPTIONS  -- None;-
        CODE-- in Ada
        begin
             if STACK.TOP=o then return true; else return false end if;
   END  --of opcs IS_EMPTY

OPERATION  Status return T_Status  is  --OPCS of POP
   DESCRIPTION Get status of class STACK code
   USED_OPERATIONS None;-
   PROPAGATED_EXCEPTIONS  -- None;-
        CODE-- in Ada
        begin
             return STACK.STATUS;
   END  --of opcs STATUS
OPERATION  POP  is  --OPCS of POP
   DESCRIPTION
   Remove an Element from the data structure
   USED_OPERATIONSNone;--
   PROPAGATED_EXCEPTIONS
        X_STACK_UNDEFINED;
   PSEUDO_CODE--
        if [lhe STACK is not EMPTY] then
             [put STACK in BUSY state]
             [remove Element from STACK_DATA]
             [put STACK in IDLE state]
        else
             raise X_EMPTY; [put STACK  in UNDEFINED state]
        end if;
   CODE-- Ada
        begin
        if STACK.TOP>0 then
             STATUS:=busy;  -- STACK code is BUSY
             Element :=STACK_DATA(STACK.TOP);
             STACK.TOP:=STACK.TOP-1;
             STATUS:=idle;-- STACK code is IDLE
        else
             STATUS:=undefined;  --STACK code  UNDEFINED
             raise X_EMPTY; --impossible case
        end if;
   END  --of opcs POP
OPERATION  PUSH is  --- code similar to that of POP
END_OBJECT STACKS
```

Figure 144 - Internals of ODS for active object STACKS

Note that this design could still be improved at the level of the OPCS code : since the operations are constrained, state conditions should only be computed within the OBCS code (and the exceptions should be raised within the OBCS[5] code); here we do the work twice!.

[5] This may lead in troubles in ADA83 because an exception raised in an Ada task and not hanlded stops a task activity; thus OBCS code should have exception handlers by default in each accept statement and at the end of the task body! IT was not made here since this makes the OBCS code highly unreadable.

2.13.2.2 Ada code generation for ACTIVE OBJECT STACKS

2.13.2.2.a Ada Specification Unit

```
with ADT_ELEMENT; -- visibility on  type T_Element   --REQUIRED OBJECTS
with PROJECT_ENV; -- visibility on  type T_Status
package STACKs is
   type T_DATA is array(integer <>) of ADT_Element.T_Element;
   type T_STACK(MAX: integer) is
   record
        DATA: T_DATA (1 .. MAX);
        TOP: integer:=1; -
        SIZE: integer:=MAX;
        STATUS: PROJECT_ENV.T_Status;
   end record;
   procedure  Status (STACK : in T_STACK) return T_Status;
   task OBCS is  -- OBCS task handling active constraint operations entries
        entry PUSH (STACK : in out T_STACK; Element : in T_Element);
        entry POP(STACK : in out T_STACK;Element : out T_Element);
   end OBCS;
   procedure  PUSH (STACK : out T_STACK; Element : in T_Element) renames OBCS.PUSH;
   procedure  POP(STACK : in T_STACK;Element : out T_Element)renames OBCS.POP;
   X_FULL : exception;  --raised by PUSH
   X_EMPTY : exception ; -- raised by POP
end STACKS;
```

Figure 145 - Ada specification Unit for active object STACKS

It has to be noted that an OBCS task has been introduced, that provides an entry for each constrained operation. Each such constrained operation is translated into a subprogram declaration that renames that entry.

2.13.2.2.b Ada Body Unit

```
with TEXT_IO;--USED OBJECTS
package body STACKS is
   function IS_FULL(STACK : in T_STACK)° return boolean is
   begin
        if STACK.TOP=STACK.SIZE then return true; else return false end if;
   end;
   function  IS_EMPTY(STACK : in T_STACK)° return booleanis
   begin
        if STACK.TOP=0 then return true; else return false end if;
   end;
   procedure OPCS_PUSH(STACK : in out T_STACK; Element : in T_Element)is
   begin-- code as entered in the code section of the OPCS-----------------------------------------
   TOP := STACK'LAST;  -- call to Attribute function
        if  TOP<STACK'RANGE then
             STACK.STATUS:=busy;  -- STACKS code now busy;
             TOP:=TOP+1;
             STACK.DATA(TOP):=Element ; --push  Element
             STACK.STATUS:=idle;-- STACKScode now idle again;
        else
             STATUS:=undefined;  --STACK code Status UNDEFINED
             raise X_FULL; --impossible case
        end if;
   end OPCS_PUSH;
   procedure  OPCS_POP (STACK : in out T_STACK;Element : out T_Element)is --similar to that of PUSH seen above

   procedure  Status (STACK : in T_STACK) return T_Status is
   begin
        return STACK.STATUS;
   end STATUS ;
   task body OBCS is separate;
end STACKS;  --end package body
```

Figure 146 - Ada Body Unit for Active STACKS

2.13.2.2.c Separated Units

```
separate (STACKS)  -- directement extrait de la section OBCS.CODE de l'ODS
        task body OBCS is
        begin
        loop -- code OBCS Ada de STACK
        select -- choice according to current state of the Stack given as parameter
              when not (IS_EMPTY) =>
              accept POP(Stack : In Out T_stack; An_Element : out T_Element) do-
                        OPCS_POP(Stack ,An_Element);         --associated OPCS body
              end POP; -- release caller thread
              or when not (IS_ FULL ) =>
              accept PUSH(Stack : In Out T_stack; An_Element : in T_Element)  do
                        OPCS_PUSH(Stack, An_Element);       -- associated OPCS body
              end PUSH;  -- release caller thread
        end select
        end loop;
        end OBCS;
OBCS body associated to constraint support
```

It can be noted that the OPCS code of the constrained operations has been put into subprograms names OPCS_*op_name t*hat have the same interface as the provided operations and which are called from within the OBCS code.
We are thus also a**chieving separation between the core functional code of the operations and the behavioural one** at code level.

Note that the implementation of the protocol constraints (HSER, LSER, ASER) is performed according to calls to the OPCS_*op_name* within or without the accept statement.

2.14 "NO TASKING" ADA CODE ILLUSTRATION

2.14.1 CODE FOR STATE CONSTRAINTS

Implementation rules for OBCS tasking code do not apply any more; instead the code of a HOOD RUN TIME library appearing as environment objects EXCEPTIONS, FSM and OBCS shall be used according to principles detailed in *Appendix A3.1* and illustrated in figure below.

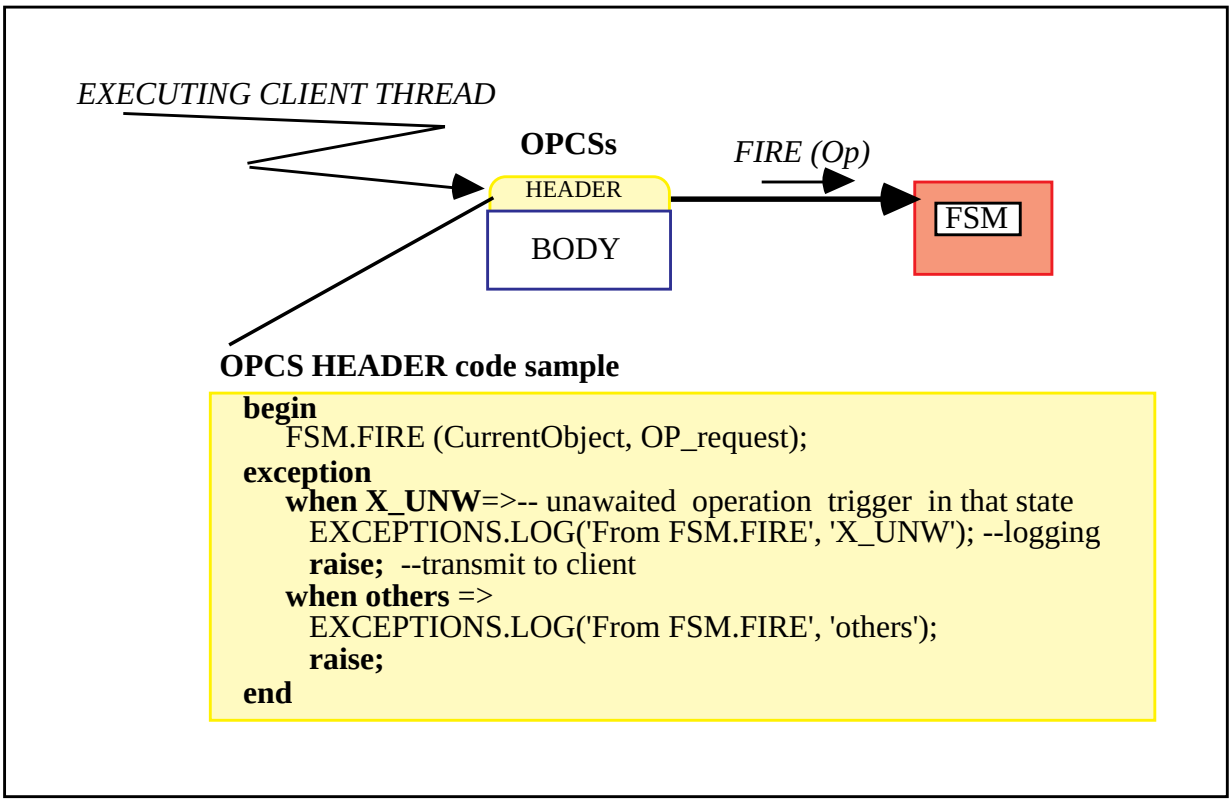


Figure 147 - Additional HEADER code to the OPCS for state constraint support with an FSM

Figure 148 - and Figure 149 - give the associated code.

```
with ADT_ELEMENT; -- visibility sur le type T_Element
with FSM; -- visibility on type T_Status defining all states of the FSM
package STACKs is
   type T_Status is (BUSY, IDLE, UNDEFINED);-- definition du type T_Status
   type T_DATA is array(integer <>) of ADT_Element.T_Element; -- useful data
   type T_STACK(MAX: integer) is
   record
         DATA: T_DATA (1 .. MAX);
         TOP: integer:=1; -
         SIZE: integer:=MAX;
         STATUS: PROJECT_ENV.T_StatuS;
   end record;
--OPERATIONS
   procedure  Status (STACK : in T_STACK) return T_Status;
--State CONSTRAINED OPERATIONS
   procedure  PUSH (STACK : out T_STACK; Element : in T_Element);
   procedure  POP(STACK : in T_STACK;Element : out T_Element);
end STACKS;
```

Figure 148 - Ada Specification Unit for State Constrained Operations without tasking

When comparing to the standard generation scheme, ones sees that the additional code to call the OBCS entries has disappeared in the specification unit. Instead we have now to reference an additional package named OSTM (Object State Machine) holding the code supporting the state transition diagram as defined in *Appendix A3.2* of present document gives a sample code for the OSTM code implementing the state transition diagram defining the behaviour of the Object as defined in Figure 140 -

```
with TEXT_IO;-USED OBJECTS
with OSTM[1]; -- acces to code handling FSM
with EXCEPTIONS -- interface for exceptions
package body STACKS is
   procedure PUSH (STACK : in out T_STACK; Element : in T_Element)is
        begin  --additional HEADER code to the OPCS body code to hanlde state conditions
             FSM.FIRE(STACKS,PUSH); -- execution request for PUSH
        exception
        when X_UNW =>
             EXCEPTIONS.LOG(" appel  FSM.FIRE(STACK,PUSH)", "X_UNW"); --logg in an file exception events
             raise;  -- transmit to caller
        when others=> EXCEPTIONS.LOG("appel  FSM.FIRE(STACK,PUSH)","others"); logg event
             raise;  -- -- transmit to caller
        end;
   ----code as entered in the CODE Section of the ODS-------------------------------------------------------------
   end PUSH;

   procedure  POP(STACK : in T_STACK;Element : out T_Element)is
   begin
        begin --additional HEADER code to the OPCS body code to hanlde state conditions
        FSM.FIRE(STACKS,POP);--execution request for  POP
        exception
        when X_UNW =>
             EXCEPTIONS.LOG(" appel  FSM.FIRE(STACK,POP)", "X_UNW");--logg in an file exception events
             raise;  -- transmit to caller
        when others=> G_EXCEPTIONS.LOG("appel  FSM.FIRE(STACK,POP)", "others");
             raise;  -- transmit to caller
        end;
        --code as entered in the CODE Section of the ODS
        end POP;

   procedure  Status (STACK : in T_STACK) return T_Status is
   begin
        return STACK.STATUS;
   end STATUS ;


end STACKS;
```

Figure 149 - Ada Body Unit for Constrained Operations without tasking

[1] OSTM stands for Object State Machine

2.14.2 CODE GENERATION FOR PROTOCOL CONSTRAINTS

Figure 151 - and Figure 152 - below give an example of Ada OBCS implementation code with no tasking according to the principles detailed in *Appendix A3.1.3* for protocol constrained operation support.

```
with ADT_ELEMENT; -- visibility sur le type T_Element
with OSTM[2]; -- acces to code handling FSM
package STACKs is
   type T_Status is (BUSY, IDLE, UNDEFINED);-- définition du type T_Status
   type T_DATA is array(integer <>) of ADT_Element.T_Element; -- données utiles
   type T_STACK(MAX: integer) is
   record
        DATA: T_DATA (1 .. MAX);
        TOP: integer:=1; -
        SIZE: integer:=MAX;
        STATUS: PROJECT_ENV.T_StatuS;
   end record;
--OPERATIONS
   procedure  STATUS (STACK : in T_STACK) return T_Status;
-- STATE AND HSER CONSTRAINED OPERATIONS
   procedure  PUSH (STACK : out T_STACK; Element : in T_Element);
   procedure  POP(STACK : in T_STACK;Element : out T_Element);
end STACKS;
```

Figure 150 - Ada Specification Unit for Constrained Operations without tasking

Note that this specification is unchanged with respect to one associated to non constrained operations. Only the ADA body unit is affected. Such translation rules allow to fully:

- separate the functional code from the behavioural one
- develop, test and integrate the "functional code", in parallel with the behavioural one.

[2] OSTM stands for Object State Machine

```
with TEXT_IO;--USED OBJECTS
with EXCEPTIONS -- visibility for  exceptions
package body STACKS is
   procedure  STATUS (STACK : in T_STACK) return T_Status is
   begin
        return STACK.STATUS;
   end STATUS ;
-- internal package  OBCS spécifique pour éviter conflits de noms avec autres objets actifs
   with EXCEPTIONS -- interface standard pour la gestion des exceptions
   with ADT_MSG ; -- Abstract Data Type defining INTER PROCESS MESSAGES ;
   package OBCS is separate;  -- see sample code below
   package body OBCS is separate;  -- see sample code belown
   procedure  PUSH (STACK : in out T_STACK; Element : in T_Element)is    --  OPCS_ER code
   MSG : ADT_MSG.T_MSG; -- local client MSG structure
   begin
        ADT_MSG.create(MSG); -- get acces to MSGs
        MSG.SENDER = STACK;  -- update MSG fields
        MSG.OPERATION:=PUSH; -- update MSG fields
        ADT_MSG.WRITE(MSG,STACK);[build_MSG] -- according to parameter list
        OBCS.INSERT(MSG); -- send in reque4st queue and wait for operation excution and return of parameters
        case  MSG.X_VALUE is  -- analysis of return parametre, possibly with exceptions
             when OK=>[Extract_MSGs]  -- according to parameter lists
             when others==>EXCEPTIONS.Raise(MSG.X_FIELD);
        end case;
        if MSG.CSTRNT/=ASER then OBCS.FREE(MSG);  --libération de la queue end if;
   exception
        when others =>EXCEPTIONS.LOG(" de Q_OP_PUSH", "Others");raise;
end PUSH;
   procedure  POP (STACK : in out T_STACK; Element : outT_Element)is--  --  OPCS_ER code
   -- code very similar to the one of POP non described here
end STACK;
```

Figure 151 - Ada body code generation with no tasking - protocol constraints

```
separate (STACK.OBCS)  --- Sample  OBCS Ada code  with noTasking
   with EXCEPTIONS --access to EXCEPTIONShandling
   with QUEUES ; --access to QUEUES  handled by the local OS
   package OBCS is --
        procedure START;  -- MAIN procedure started at initialisation
        procedure INSERT (MSG:in T_MSG);
        procedure INIT_POOL(MAX : integer:=10); -- défaut = 10 pending calls
        procedure ALLOC(MSG:in T_MSG);
        procedure FREE (MSG:in T_MSG);
   end OBCS;

package body OBCS is  --
   type T_Pool_Status is (FREE, ALLOCATED);--                --local types internal to the OBCS
   type T_Data is array(integer <>) of QUEUES.T_QUEUE;  -- environmental QUEING library
   type T_Pool(MAX: integer:=10) is recordData : T_Data (1..MAX);
                                          Status : T_Pool_Status:=FREE;
                                    end record;
   POOLMAX : constant integer:=10;  -- 10 pending calls;
   -- local data
   R_Q: QUEUES.T_QUEUE;  -- execution request queue
   POOL : T_POOL(POOLMAX);

   procedure Init_pool(Max : in integer:=10) is
   begin
        for i in 1 .. MAX loop
             QUEUES.INITIALIZE(Pool.DATA(i), 1); --
             Pool.status(i):=FREE;
        end loop;
   end init_pool;
   procedure ALLOC (MSG:in out T_MSG) is
   begin
        for i in 1 .. POOLMAX loop
        if Pool.Status(i):=FREE then
             Pool.Status(i):=ALLOCATED; MSG.RTN_Q:=Pool.data(i);return;
        end loop;
        raise X_OBCS_NOMORE_RTN_QUEUES;
   end ALLOC;
   procedure FREE (MSG : in T_MSG) is --FREE a QUEUE
   begin
   for i in 1 .. POOLMAX loop
        if Pool.Data(i):=MSG.RTN_Q then Pool.Status(i):=FREE; return; end if;
        end loop;
        raise X_OBCS_FREE_INCONSISTENT;
   end FREE;

   procedure INSERT (MSG: in T_MSG) is --insert Requests
   begin
        QUEUES.Insert(R_Q,MSG);  -- insert in Request QUEUE
        OBCS.ALLOC(MSG) -- allocate a RETURN QUEUE access into MSG
        QUEUES.Remove(MSG.RTN_Q,MSG); -- wait on return of operation executionr
   exception   when others =>EXCEPTIONS.LOG(" de OBCS.INSERT", "Others");raise;
   end INSERT;

   procedure SerDispatcher(Name : T_Operation)  is separate;  -- dedicated procedure for each object
```

```
   procedure START is  -- MAIN OBCSServer process
   MSG : ADT_MSG.T_MSG; -- local MSG data
   begin
   OBCS.INIT_POOL ;-- init of QUEUE pool
   loop
         QUEUES.Remove(R_Q,MSG);  --get a request from R_Q Queue
          case MSG.CSTRNT is  --process the MSG
               when  HSER| HSER_TOER =>
                     SerDispatcher(MSG.OPERATION); -- execution du code concerné
                     QUEUES.Insert(MSG.RTN_Q,MSG); -- renvoi MSG au client
               when LSER |LSER_TOER=>
                     QUEUES.Insert(MSG.RTN_Q,MSG); -- renvoi MSG au client
                     SerDispatcher(MSG.OPERATION); -- execution du code concerné
               when ASER => SerDispatcher(MSG.OPERATION); -- execution du code concerné
               when others =>EXCEPTIONS.LOG(" STACK OBCS INIT PROCESS LOOP", "BAD constraints")
                     raise X_MSG_INCONSISTENCY;
         end case;
   end loop;
   exception  when others => EXCEPTIONS.LOG(" STACK OBCS INIT PROCESS LOOP", "EXCEPTION.NAME");
end START;
end OBCS;
```

Figure 152 - OBCS body code with no tasking - protocol constraints

```
separate (STACK.OBCS) -

procedure  OPCS_POP(STACK : in T_STACK;Element : out T_Element)is separate;--effective  STACK_server code
procedure  OPCS_PUSH(STACK : in T_STACK;Element : out T_Element)is separate;

   procedure SerDispatcher(Name : T_Operation)  is \---
-------------------------------------------------------------------------------------------------------------------------------------------------------------
-- if we want CLIENT_SERVER code, then we would have to suppress the STACK object in the parametre list and
declare it here as a local data of the server
   --Local _STACK: T_STACK; -- local Data for client-server code
-- such code generation schema could be triggered by a pragma--
-------------------------------------------------------------------------------------------------------------------------------------------------------------
         Local_Element : ADT_ELMENT.T_ELEMENT;
         begin
   case Name is  --
         when PUSH=>begin
               ADT_MSG.READ(MSG,Local_STACK);--[Extract_MSG] unmarshall parametres
               ADT_MSG.READ(MSG,Local_Element);
               Opcs_PUSH(Local_STACK,Local_Element); --effective local server execution of the operation
               MSG.X_VALUE=OK; -- set MSG return MSG info
               exception
               when •••=>MSG.X.VALUE=X_•••
               when X_UNW=> MSG.X_VALUE=X_UNW;
               when others=>EXCEPTIONS.LOG("SerDispatcher, OPCS_PUSH", "Others");
                              MSG.X_VALUE=X_OTHERS; end ;
         when POP=>begin
               ADT_MSG.READ(MSG,Local_STACK);--[Extract_MSG]sunmarshall parametres
               Opcs_POP(Local_STACK,Local_Element);--appel effectif
               ADT_MSG.WRITE(MSG,Local_lement)---[build_MSG]marshall parametress
               MSG.X_VALUE=OK;
               exception
               when X_UNW=> MSG.X_VALUE=X_UNW;
               when others=>EXCEPTIONS.LOG("SerDispatcher, OPCS_POP", "Others");
                              MSG.X_VALUE=X_OTHERS; end;
         when others => EXCEPTIONS.LOG("  OBCS.OPCS.NAME", "NO SUCH NAME");
   end case;
   exception
         when others => EXCEPTIONS.LOG(" OBCS.OPCS.NAME", "Others");raise:
   end SerDispatcher;
```

Figure 153 - Separate part of OBCS body code containing the dedicated SerDispatcher code

```
separate (STACK.OBCS.SerDispatcher)  -- STACK server code

   procedure  OPCS_POP(STACK : in T_STACK;Element : out T_Element)is
        -- same code as Figure 149 -
   end OPCS_POP;

   procedure  OPCS_PUSH (STACK : in out T_STACK; Element : in T_Element)is
        -- same code as Figure 149 -
   end OPCS_PUSH;
```

We can see that the principle of translating the code of protocol constrained operation OPCS in to OPCS_*op_Name* still holds, in that most of the code OPCS_ER code can still be automatically generated.

The OBCS implementation may slightly vary according to queuing inter-process queuing mechanisms available for the target.

3 GETTING THROUGH

3.1 HANDLING DOCUMENTATION STANDARDS

The **HOOD method does not mandate any documentation standard** neither for Architectural Design Document (ADD) nor for Interface Control Document (ICD) nor for Detailed Design Document (DDD).Two cases may take place:

- **no mandated documentation standard for the project**: the design documentation is simply defined around the ODS concept as **a set of ODSs**. An ODS is unique in a document[1]. A design document is structured according to
  - the three spaces of hierarchies; objects, classes and Virtual Nodes
  - the parent-child relationships associated to each hierarchy
- **a documentation standard for the project is mandated** (DO2167, ESA...); the standard documentation is produced from the HOOD one**,** and is defined according to the following principles:
  - Extraction of documentation part from ODS fields
  - **Tuning of the standard** (in accordance with prime QA team) in order to avoid **redundancies** in the documentation
    Implementation of such approaches on industrial projects already exist (see HOOD and DOD2167 standards [SEXTANT92]).

In order to build a complete documentation as defined by a standard[2] it is necessary to add information to the ODS, during the HOOD design process (such as “Justification of Design Choices“ part). That information may generally not be included in the ODS and HOOD tools handle it differently.Each type of document (ADD, ICD, DDD) needs so to extract a sub-set of HOOD information included or not in the ODS, related to the life-cycle phase to be documented.

A design document (particularly ADD) needs to explain the design and does not necessary include all information coming from the design process. For example an ADD does not include the identification of nouns and verbs from the informal strategy text. In the same way, the process defining a method to build the system first through the logical architecture and then through the physical one, is also not present in the ADD. The ADD is seen as a result of this process. If needed, intermediate results of the process may be traced in specific technical notes or in the sections allowing to justify particular design choices.

Redundancies may be produced to ease readability of documentation. Those redundancies shall be generated by tools in order to keep consistency. For example it could be interesting to print an operation description in all places where that operation is referenced (Provided interface of the server object and all Required interfaces of client objects).

At last, even if all objects are not completely identified across the architectural design phase, we recommend to build an ADD which includes all objects and hierarchies. Thus, the ADD may simply be updated after the detailed design phase in order to reflect the final whole system de-

[1] Otherwise a reader would be confused!

[2] This chapter does not describe parts of the design not covered by the HOOD method such as Database tables description, Dynamic architecture, etc...

sign.note also that a DDD should only include pseudo code and internal description of terminal objects.
The following sections provide standards for each type of document. These are only examples which will be customised according to the projects needs or standards, or even company internal standards (ESA PSS-05, DOD 2167a...).

3.1.1 ADD STANDARD

3.1.1.1 Introduction

Before building a document standard, it is necessary to think about the reader. For an ADD, many kinds of readers may have to review it: the specifier, the reviewer, the designer, the developer, the maintainer... All those people need different design information. For example, a specifier may be more interested by the global organisation and external interfaces even though a developer is interested by the complete ODS he has to develop.
Hence, we suggest to organise the ADD in order to answer to all those people: an ODS may be put in appendix of the ADD and may not be provided to all readers. Nevertheless, it is always possible to introduce ODS directly at the level where an object is defined (i.e. at the end of the related section).

An ADD covers the whole system configuration or a part of it. If the ADD is related to a system, it will includes the VNTs and all HDTs (and the related CDTs) allocated into terminal virtual nodes. If an ADD is related to an object of a HDT, it will only describe the decomposition of that object and the used environment objects.

Finally, in order to build a readable ADD, a table of contents is fundamental to retrieve objects. Moreover, we recommend to add an index at the end of the ADD allowing to list objects, types, operations and constants with, for each of them, pages where the element is defined and used. This index will be very useful for integration phase and maintenance activity.

3.1.1.2 Example of ADD Contents

1. System/Software General Overview
1.1 Main Functions
This section introduces the main functions of the system or software to be designed, its aim and mission.
1.2 Overall Organisation
This section describes the general system organisation and gives some justification on it.
1.2.1 System Configuration
This section introduces all the design trees used to define the whole system (VNT, HDTs, CDTs) and links between them.
1.2.2. Applicative Environment
This section describes the HDTs and CDTs defining the Applicative environment.
1.2.3. Implementation Environment
This section describes the HDTs and CDTs defining the Implementation environment.
1.2.4. Justification of Organisation
This section gives some explanation about the project organisation.
1.3 Development Constraints
This section gives some particular development rules or constraints (i.e. the language, a real-time monitor...).
1.4 HOOD Tool
This section includes the used HOOD tool presentation.

A chapter (2, X and Y) is dedicated to each design tree (from the VNT to the HDTs and the CDTs) covered by the ADD:

2. <*System*> Design-- *For the Virtual Node (if any) describing the system to be designed*
This chapter describes the Virtual Node Tree (VNT) of the system.
2.1. Overview
2.2. Hardware/Physical Environment
This section defines the operational physical environment of the system to be designed in terms of processors, stations, etc... on which the system will run.
2.3. Resources Allocation
This section defines the allocation of the Virtual Nodes to the Physical Nodes.
2.4. Software Environment
This section defines the operational software environment, that means all software in communication with the system to be designed.
2.4.1. Overview
2.4.2. Functioning
This section explains the following graphical description in terms of data handling through operations of the different virtual nodes.
2.4.3. Graphical Description
This section gives a context diagram of the root virtual node defining the system to be designed. It is defined with a HOOD diagram including the system virtual node and all other virtual nodes communicating with the system.
2.5. Dynamic Architecture
This section describes general principles of the dynamic architecture such as time management, tasks or process organisation and synchronisation between them, performance principles, software starting, etc...
2.6. Static Design
This section gives the static decomposition of the system first in virtual nodes and then into objects allocated in the terminal virtual nodes.
2.6.1. Overview
2.6.2. Design Tree
This section gives a synthetic view of the virtual nodes hierarchy from the root until allocated objects.
For each Virtual Node or allocated Object of the Virtual Node Tree:
2.6.x. <*Virtual Node*> Virtual Node | <*Object*> Object
2.6.x.1. Overview
This section gives a presentation of the virtual node (or object) corresponding to the related ODS description part. When virtual node, it could be an output of the “Statement of the Problem“ part of the HOOD design step.
It may also include a synthesis of the requirements covered by that virtual node (or object). In that sense, it is an output of the ”Analysis and Structuring of the Requirements Data“ part of the HOOD design step.
When virtual node, it may also include particular design principles followed in the decomposition of that virtual node.
2.6.x.2. Functioning-- *Only for Virtual Node*
This section explains the following graphical description in terms of data handling through operations of the different child virtual nodes or allocated objects. It is an output of the “Informal Solution Strategy“ part of the HOOD design step.
2.6.x.3. Graphical Description-- *Only for Virtual Node*
This section gives the HOOD diagram of the current virtual node defining the decomposition of that node into virtual nodes or into allocated objects.
2.6.x.4. Justification of Design Choices-- *Only for Virtual Node*
This section (facultative) includes the set of design decisions made for the current decomposition. It may include reason of particular choices, rejected choices, etc... It is an output of the “Justification of Design Decisions“ part of the HOOD design step.
-- When object is an instance of a class it could be interesting to explain the reason of that choice (reusability, similar objects...).

For each decomposed Object allocated in the previous Virtual Node Tree or the Object to be designed:

X. <*Object*> Object Design
This chapter describes the Design Tree of the object.
X.1. Overview
X.2. Environment
This section defines the object environment, that means all environment objects used by the object to be designed.
X.2.1. Overview
X 2.2. Functioning
This section explains the following graphical description in terms of data handling through operations of the different objects.
X.2.3. Graphical Description
This section gives a context diagram of the root object. It is defined with a HOOD diagram including the object to be designed and all used environment objects.
X.3. Static Design
This section gives the static decomposition of the object into child objects (include relationship) until terminal objects.
X.3.1. Overview
X.3.2. Design Tree
This section gives a synthetic view of the objects hierarchy from the root object until terminal objects.For each Object or Object Instance of the Design Tree:

X.3.x. <*Object*> Object
X.3.x.1. Overview
This section gives a presentation of the current object corresponding to the related ODS description part. When non terminal object, it could be an output of the “Statement of the Problem“ part of the HOOD design step.
-- It may also include a synthesis of the requirements covered by that object. In that sense, it is an output of the ”Analysis and Structuring of the Requirements Data“ part of the HOOD design step.
-- When non terminal object, it may also include particular design principles followed in the decomposition of that object (such as portability, reusability, etc...).
X.3.x.2. Functioning-- *Only for Non Terminal Object*
This section explains the following graphical description in terms of data handling through operations of the different child objects. It is an output of the “Informal Solution Strategy“ part of the HOOD design step.
X.3.x.3. Graphical Description-- *Only for Non Terminal Object*
This section gives the HOOD diagram of the current object defining the decomposition of it into child objects.
X.3.x.4. Justification of Design Choices-- *Only for Non Terminal Object*
-- See above (2.6.x.4)

For each non decomposed object of the Virtual Node Tree or Environment object of the previous design trees:

Y. <*Object*> Environment Object
Y.1. Overview
This part explains the reason why that object is external to the system to be designed.
It may also include a synthesis of the requirements covered by that object.
Y.2. Object Description
This section gives the ODS of the current environment object.

For each Class instanciated into the Design or Class Trees:

Y. <*Class*> Class Design

This chapter describes the Design Tree of the class.

Y.1. Overview

Y.2. Environment

This section defines the class environment, that means all environment objects used by the class to be designed.

Y.2.1. Overview

Y.2.2. Functioning

This section explains the following graphical description in terms of data handling through operations of the class and the used environment objects.

Y.2.3. Graphical Description

This section gives a context diagram of the class. It is defined with a HOOD diagram including the class to be designed and all used environment objects.

Y.3. Static Design

This section gives the static decomposition of the class into child objects (include relationship) until terminal objects.

Y.3.1. Overview

Y.3.2. Design Tree

This section gives a synthetic view of the class hierarchy from the class until terminal objects.

For each class, object or object instance of the Design Tree:

Y.4.x. <*Object*> Object

X.4.y.1. Overview

-- See above (X.3.x.1).

Y.4.x.2. Functioning-- *Only for Non Terminal Object*

-- See above (X.3.x.2).

Y.4.x.3. Graphical Description-- *Only for Non Terminal Object*

This section gives the HOOD diagram of the current object defining the decomposition of it into child objects. It particularly highlights use links with formal parameters.

Y.4.x.4. Justification of Design Choices-- *Only for Non Terminal Object*

-- See above (2.6.x.4).

APPENDIX A. Textual Descriptions of Objects

For each Virtual Node, Object or Class defined in the previous chapters, this appendix gives the related ODS. We suggest to classify all those ODS in alphabetic order.

A.x. <*Virtual Node*> Virtual Node ODS | <*Object*> Object ODS | <*Class*> Class ODS

-- ODS are complete for non terminal objects, virtual node and class.

-- ODS for terminal objects and OP_control only include the visible part. Indeed, the ADD does not include information about pseudo code and code.

APPENDIX B. Traceability Matrix with SRD

This matrix traces each requirement of the SRD by an object of the design.

3.1.2 DDD STANDARD

3.1.2.1 Introduction

According to the suggested standard one DDD is required for the whole ADD or for each object of the ADD. For simplicity reasons, we recommend to have one DDD for each ADD.

3.1.2.2 Example of DDD Contents

1. System/Software General Overview
This chapter introduces the main functions of the system or software to be designed, its aim and mission. It may also include a HOOD tool presentation and give particular development constraints. See section 3.1.1.2.

For each hierarchy (HDTs and CDTs) defined in the ADD:

X. <*Root Object*> Design Tree
This chapter includes all ODS of terminal objects **and OP_control**. ODS of other objects are of no interest for they are described in the ADD and redundancies between those two documents is not recommended.

X.1 Design Tree
This section gives an overview of the current object decomposition. This representation allows to easily identify location of objects.

For each terminal object or OP_control of the current design tree (may by classified in alphabetic order):

X.x. <*Object*> ODS
-- OD Sshall not include code fields for OPCS and OBCS.

3.2 MAINTENAINING CONSISTENCY BETWEEN DESIGN AND CODE

A particular problem of consistency arises from descriptions elaborated during detailed design phase within the HOOD ODS and the generated target code units, which may have to be modified after generation according to debugging and target specific constraints.

3.2.1 PROBLEM DEFINITION

HOOD3.1 defines ODS CODE FIELDS that are copied into an architecture of target code units skeletons, and that are correctly related according to HOOD required and provided interfaces relationships. When target code is generated from a full system configuration two cases may happen:

- the generated code is correct and compiles. - fine
- the generated code is not correct, and must be modified. Either is it due to syntactic errors, or has the code to be adapted to the specific target, or it is due to an error in the code generation from the toolset.

3.2.2 MASTERING THE RELATIONSHIP BETWEEN ODS AND CODE FILES

According to our experience on earlier HOOD projects, where a complete separation was enforced between code and the ODS descriptions, we think that ODS code descriptions fields must be managed in configuration together with the final code. Otherwise, a gap always exists between HOOD ODS descriptions and the final code, and developers tend to rely only on the code: *consequently the goal of HOOD to achieve reliable high level descriptions of less complexity and volume than code is missed.*

We may distinguish two types of code:

- Code corresponding to architectural concepts (such as modules (in C) or packages (in Ada), WITH clause (in Ada) or Include (in C), provided types, constants, etc...),
- Code defining control structures of operations, algorithms.., corresponding to detailed design and coding activities.

Any modification on the first point relies from an Architectural Design activity and modification of operations internal code relies from a Coding activity.
Thus, an important rule is that modification performed directly in the source code shall not impact the design (no modification at the interface level for example).

Two methods may be defined for handling code consistency with design descriptions. Each of them requires particular procedures as detailed in sections below.

3.2.2.1 Maintaining Code definition within the ODS:

When the development is terminating (end of integration or unit testing) and before putting the product under configuration management and **before** each design modification:

- Generate code from ODSs with the HOOD toolset and save previous code versions,
- Modify that code (only the one associated to ODS internal fields), during coding and unit testing, using the full power of usual development environment (code editors, browsers, debuggers).
- Update terminal ODS fields from achieved actual target code.
  Such update requires using particular sofwatre which can defined:
  - using UNIX tools such as *diff* and *sed* and *awk* for example.
  - using a dedicated software program and commenting rules that would analyze the fianlized target code, recognize code include from ods fields, and rewite that code back in the HOOD ODSs database. Such a practice was used in several projects using a HOOD tool having an simple ODS database schema based on UNIX file system.
- In case target code modificationswould have impacted the design architecture,two updates must be performed (one before the other according to the scope of the modifications[3]:
  - updating the architecture by updateing the ODS «provided fields»
  - updating the code fields of the ODS
- Set the ODS under configuration.

3.2.2.2 Maintaining consistency between Code and ODSs

When the development is terminating (end of integration or unit testing) and before putting the product under configuration management and **after** a design modification:

- Generate code from ODSs with the HOOD toolset and save previous code versions,
- Modify that code (only the one associated to ODS internal fields), during coding and unit testing, using the full power of usual development environment (code editors, browsers, debuggers).
- Update terminal ODS fields from achieved actual targt code.
  Such update requires using particular sofwatre which can defined:
  - using UNIX tools such as *diff* and *sed* and *awk* for example.
  - using a dedicated software program and commenting rules that would analyze the fianlized target code, recognize code include from ods fields, and rewite that code back in the HOOD ODSs database. Such a practice was used in several projects using a HOOD tool having an simple ODS database schema based on UNIX file system.
- Set the ODS under configuration.

[3]. for example if an operation has been added in an HOOD object, one should firts modify the design, and then the ODS code field updates.

3.3 REUSING ENVIRONMENTAL SOFTWARE (NON HOOD-CODE)

Environmental code should be specified (when used from within a HOOD design tree) as HOOD Environments objects, what requires modelling all resources used or reused as provided items of such a HOOD object. A problem might occur if one has to reuse “global data” since those data are not allowed in provided interfaces of HOOD objects: the solution is then simply to model such data as a HOOD object with two operations (readata and writedata).
A HOOD toolset will thus ne able to check interfaces with reused code, and the effective module may simply replace the associated “body part” generated by the HOOD toolset.

3.4 MANAGING HOOD PROJECTS

3.4.1 OVERVIEW

HOOD is a design method allowing to produce modular and hierarchical descriptions of software, as a set of interconnected objects. The specifics of a HOOD development is that the refinement of such a model is performed along through two kind of activities:

- **refinement by adding more details to object descriptions**: the activity proceeds by enriching the descriptions of an object by adding more details to an existing description, using "step-wise refinement techniques" within the INTERNALS framework of an ODS (Object Description Skeleton).
- **refinement by decomposition**: according to the properties of the include relationship, a HOOD model can be refined by adding more objects to an initial model. Thus the description of *the initial model is left unchanged during the remaining refinement activity.*

This concept of initial invariant model development is fundamental and specific to HOOD: a development step will thus be a step allowing to freeze, prototype and validate a model refinement, still consistent and equivalent to a initial one. Figure 154 - and Figure 155 -hereafter illustrate this process with:

- a first level of description elaborated at a date say T1, is composed of three objects and represented by a design tree (that's 4 objects)

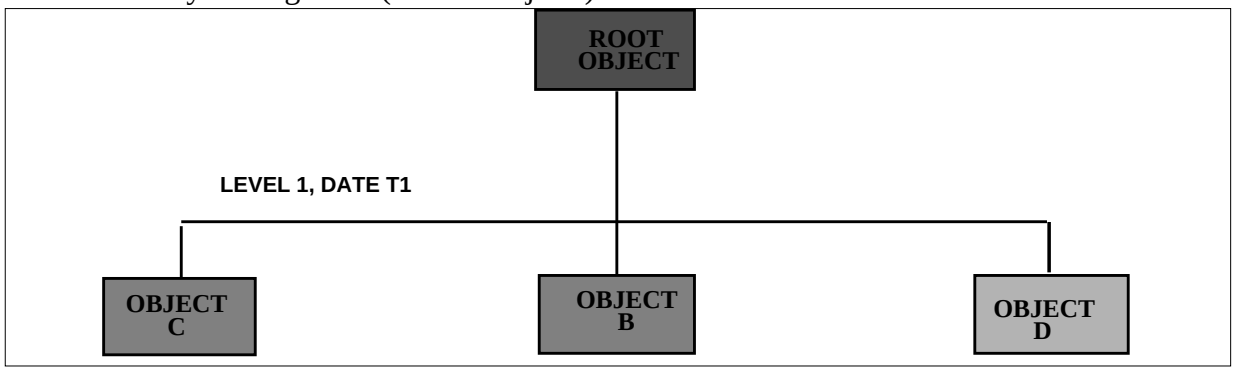


Figure 154 - Example of a HOOD initial model at level1 of decomposition

- a second level of description more refined and detailed in Figure 155 -, elaborated at a date T2 (T2>T1) is described as design tree with 9 objects.

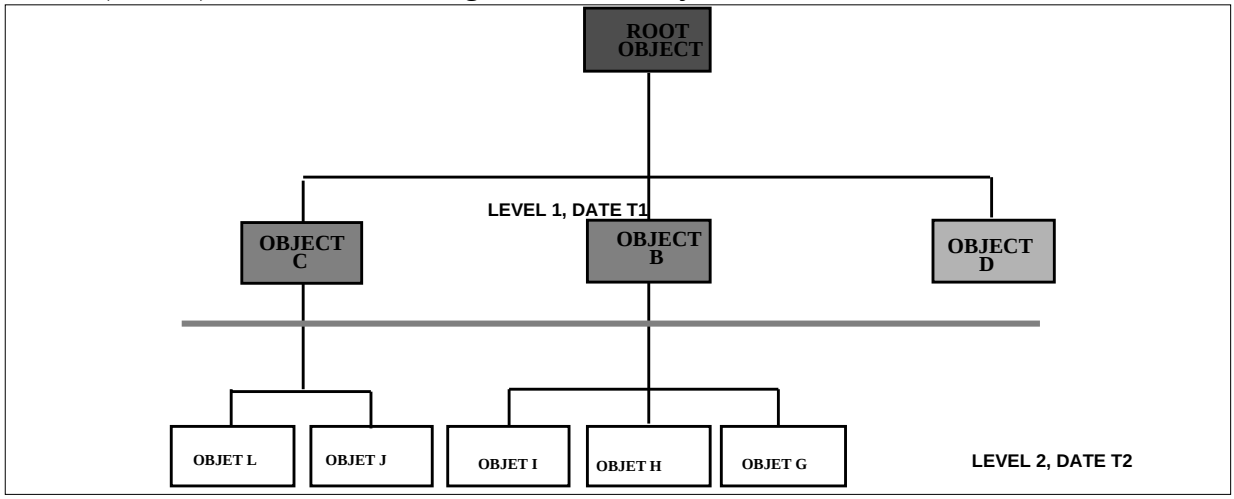


Figure 155 - Example of Refinement of initial HOOD model

Putting into work such an approach for a full project leads into technical activities organized according to an incremental life-cycle, whereas current development standards generally consider only the classical waterfall model or “V” life-cycle. Hence definition of architectural and detailed design activities became difficult with HOOD.

A first-shot technical categorization of design activities following HOOD properties may however be achieved in considering the object as a unit of configuration, thus:

- **architectural design**: is a set of refinement activities by decomposition.
- **detailed design**: is a set of refinement activities by enrichment of descriptions of terminal.objects using step-wise refinement on pseudo-code and/or code descriptions. (A parent object is fully defined by its children. If all children are specified, then the parent is also defined).

However, the definition of an architectural design review (ADR) still has to apply to a model where the architectural definition choices have been made and been validated. ADR should only be applied to significant HOOD models, i.e. where some validation has been performed.

Moreover **after a ADR, more objects may still be added,** even this leads to an update of the ADD, an possibly to another treview of the ADD!

Figure 156 - hereafter illustrates several design trees associated to a HOOD development, and a “significant one” l reviewed at ADR.

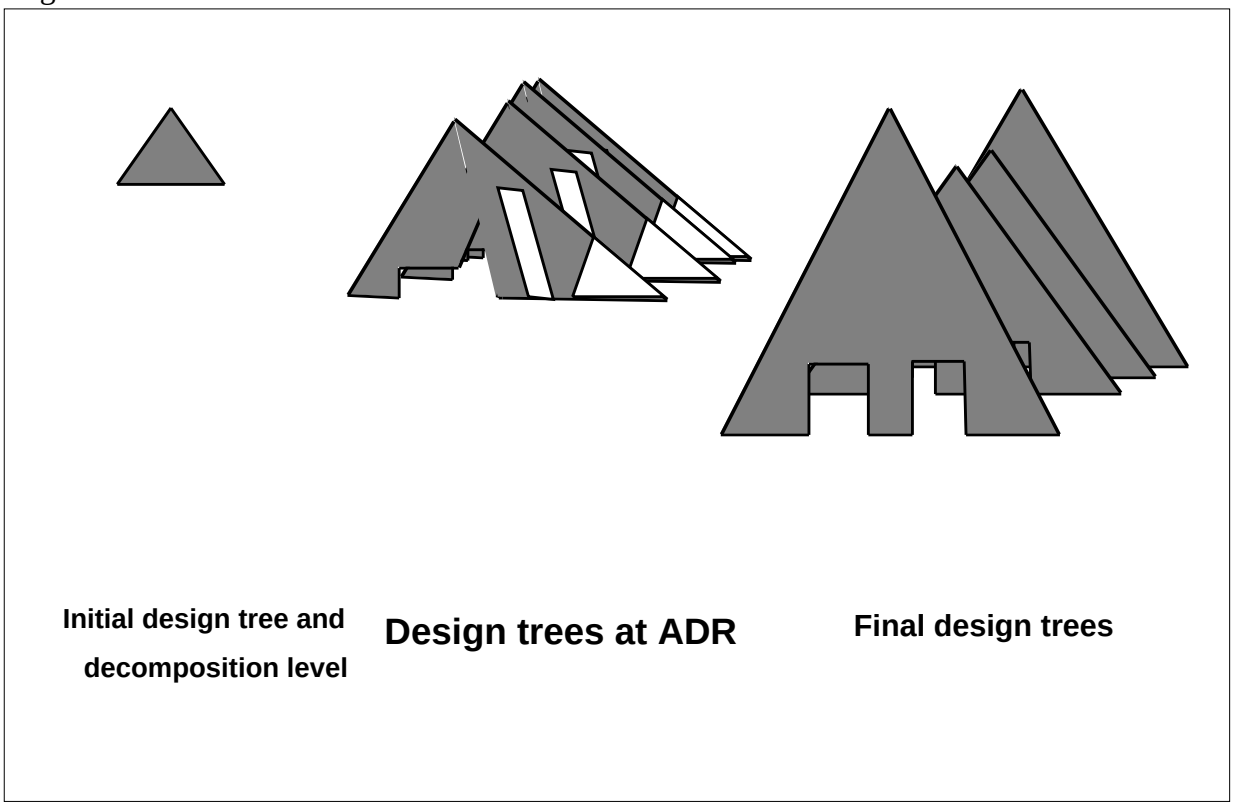


Figure 156 - HOOD Design trees states in the development life-cycle

3.4.2 SUBCONTRACTING

As development activities associated to a HOOD development will be spread over several development teams, the management of these activities shall be tuned, and adapted to the HOOD models and development approach .
The process of defining the work breakdown and the allocation to subcontractors is complex, depends on multiple factors, as well as on the industrial organisation defined to support a project. However the task definition work will generally include one or several of the following activities:

- **Elaboration of a HOOD initial model.** Child objects of the top-level system define as much subsystems that can be developed in parallel. Elaboration of the HOOD initial model may correspond itself to a sub project and depending of the validation effort, may consume from 30% to 40% of the project resources.
- **Definition of HOOD system_configurations associated to subcontracted objects.** The development contexts of **e**ach subcontractor are defined, enforcing whenever needed confidentiality constraints.
- **Definition of the VN architecture.** This task is performed either in parallel or in the follow-up of the elaboration of the initial model. The architecture which is set-up (and possibly prototyped) should be compliant in terms of performances, target system, and eventually to a domain application generic model (extracted through capitalisation of the know how in the domain).
- **Allocation of objects onto Vns**. This task allows grouping objects of the initial model according to physical and/or organisational constraints.
- **Elaboration of associated technical Requirement Specifications(STB)** eventually extracting information from the HOOD documentation associated with the initial model
- **Definition and contractual allocation of development tasks for subcontractors.** according to the work breakdown established above.
- **Level validation allowing development factorisation of common resources through formal reviews.** Such reviews are only mandatory for developments that produce new objects in their refinements.
- **Pre-integration and integration of object/subsystems subcontracted.** Subcontractors can start pre-integration on their own sites, if they can access to the prototypes developed by their prime.(on its own site or not)
- **Final Integration and validation.** This task is mandatory performed on the prime site

During all the development, consistency and configuration management of the global HOOD model can nevetheless be ensured thanks to the concepts of environment objects and system-configurations. Two concepts shall be distinguished: (see Figure 157 - below)

- the global *system_configuration of the project at prime level.* Such a configuration shall be handled at HOOD level and shall define the configuration of the project by integrating all object and class hierarchies.
- the local *system_configuration of a subcontractor.* Such a configuration is at least the one defining the context of the hierarchy associated to the local development. The subcontractor will enrich it with new environment objects and classes, as he progress in refinement his object/subsystem hierarchies.

3.4.2.1 Allocation of objects to subcontractors

Allocating objects to subcontractors shall make trade-offs between constraints induced by:

- a fine breakdown of the technical model at several decomposition levels. Such a breakdown should not have a too fine granularity (there are serious risks of doing the work of the subcontractors) but at the same time should be significant enough to allow a trusty evaluation of the effort, and further allocation without impacting changes to the initial model.
- the allocation of associated resources for the definition of the breakdown .

The more the initial breakdown is fine, the more the resources necessary. Moreover the allocation of resources is often made, before the effort needed to do the system is really known!!!

3.4.2.2 Managing the consistency

The prime shall foresee a number of reviews in order to synchronise the parallel developments by its subcontractors. The system configuration of the subcontractors will be updated at the time of these reviews, since root objects of each local system configuration are likely to be common resources at the global system configuration level.
Thus, early identification of common resources is a matter of efficiency and of resource optimization. Figure 157 - below tries to illustrate the relationships between local and global system_configuration.

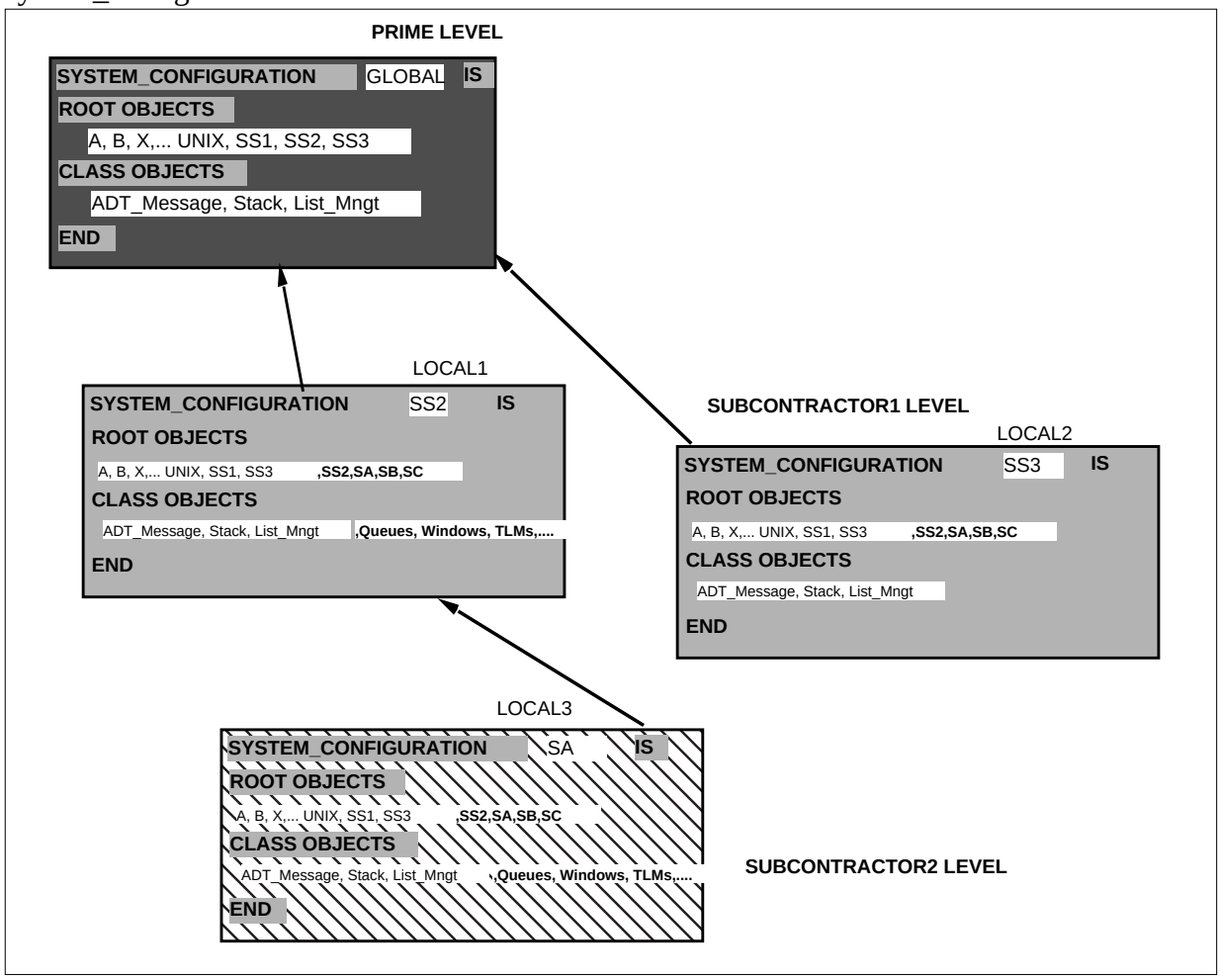


Figure 157 - Consistency of multi System-Configurations

3.4.3 CONFIGURATION MANAGEMENT

3.4.3.1 Principles

The principles of configuration management are to master the definition of components of a product. Within a HOOD development, the ODS is the primary unit of configuration and, at any moment, the configuration of a HOOD development is defined by the associated ODSs.

If we admit that the code is generated directly from a HOOD toolset, (with possibly a few additional modification procedures), all additional procedures (*shell/sed/make*) for building the executable programs should also be configuration items.

Moreover we have seen that the “SYSTEM_CONFIGURATION” must be associated to a given hierarchy of objects, hence the “SYSTEM_CONFIGURATION” is also a configuration item.

Finally the choice of the granularity of the configuration items (of lower granularity than the ODS or not) will depend of the constraints and means of a project.

In all what follows and to simplify, we illustrate this principles with the hypothesis of the ODS as the basic unit of configuration.

3.4.3.2 Configuration of a Development

The ODS is a set of text sections that contain all information needed by the developer, including the code. One can define specific development states according to two point of views:. technical and organisational.

- From the technical point of view the development states correspond to the three states of ODS: CHILD state, PARENT state and TERMINAL state as defined in section 2.2.4.
- From the organisational point of view, the states correspond to project milestones: the end of the architectural design phase, the end of the detailed design phase, the end of the coding phase.

The first configuration of a project may be defined at the end of the architectural design phase; it corresponds to the definition (and optionally to the prototyping) of an initial HOOD model defining a contractual reference state. From this configuration on, test specification and plan will be derived as well as traceability activities.
Such a configuration is defined asone system configuration defining a set of objects and class hierarchies and described by:

- ODSs in state CHILD
- ODSs in state PARENT
- possibly some ODS in state TERMINAL, (namely those associated to a prototype;).

3.4.3.3 Configuration Management of Code- versus -ODS

HOOD3.1 integrates in the ODSs CODE description fields, which are “copied as is” in target unit modules. When code is generated from the ODS information two cases may take place:

- code is correct
- code is incorrect :
  - either because of a bug in the code generator of the HOOD toolsets used,
  - either because the developer had to bring modifications in the generated code to adapt it to particular target constraints (add of Ada pragmas such as pragma in-line, SUPPRESS_CHECKS, representation clauses, etc..).

From the feedback of early HOOD projects where there was a separation between ODS and target code, we think that ODSs must handled in configuration with the target code.This can be implemented by adding modification procedures upon generated code(MPUGC). Such MPUGCs may be stored as additional description fields of a given ODS (pragma HOOD ODS) (e.g as a subset of the “IMPLEMENTATION_CONSTRAINTS” field). As a result additional operational constraints are brought to the project :

- Code generation from the ODS using a standard HOOD toolset
  For each object generated code is modified by executing the modification procedure MPUGC(if any) extracted from the ODS,
- Target Code Evolution.
  During coding and unit testing this code may evolve as the developer uses the full features of the development environment (Editiors, Browsers, Compilers, Debuggers, TestBenchesetc). However it will have to keep a list of all modifications he made on the generated code
- Archive and [re]setting configuration.
  The developer must now update the target code (and possibly associated generation procedures) according to the list of traet code modifications since the last stable state. For this he will (either manually or using a dedicated tool):
  - Update CODE fields of ODSs.
  - Update code modification procedures. Such procedures may be a set of the batch editor commands such as SED under UNIX environments.
  - Regenerate for validation.

3.4.3.4 Evolution of the Project Configurations

A configuration definition at the end of the detailed design does not seem to be needed: the evolution with respect to the first configuration definition may be performed in terms of added HOOD objects and states (transitions from CHILD state to TERMINAL state).

Moreover experience showed that the state obtained at the end of the detailed design is often not to much stable, what has also leaded to the suppression of formal Detailed Design Review.

We suggest rather to handle a second configuration, at the end of the coding and unit testing phase. Such configuration may be thoroughly reviewed (inspections, authors-reader cycles) and more formally by automated generation of target code.
From that point only , the design will be put under change control and every code modification shall lead to a change within an ODS, and a regeneration, with replay of test to validate the modification.

3.4.4 REVIEWING A HOOD DESIGN

3.4.4.1 Warning

When one has to evaluate a design one may face with two figures:

- All verification et validation recommended by HOOD, as described in section 2.3, wererегularly performed during the development and can be traced through validation reports and minutes.
  In this case, a documentation is provided allowing to check that the solution works properly and implements the requirements. Such documentation can be summarized as:
  - a description of the system_configuration
  - for each hierarchy of the system_configuration:
    - the design tree
    - for each object or class its description (sections H1, H2, H3), its provided interface, its dataflows and its behaviour.
    - a validation report about verification along the design process

- Only a minimal verifications have been preformed, or no validation trace is available.
  In this second case, all validation activities have to be done as the design is evaluated.

In the following we set up such an hypothesis. Hence all validation activities must be performed and the documentation delivered must be a full documentation.

3.4.4.2 Documentation Structure

Evaluation a design goes through the reading of ODSs describing fully the architecture. The organisation and layout of these ODSs is important and should try to ease the reading and the understanding of the design.

The set of ODSs associated to a design is generally presented as a top-level hierarchy of objects.If the designer has chosen to present the design tree in a vertical way, branch after branch, one may find the ODS of a child of the root object of the system-to-design at page 3000 of the document.! Hence we recommend to have a presentation of ODSs following an horizontal organisation, level by level.

Furthermore it is important that in a same document submitted to a review, an ODS appears only once, otherwise the reader would be very confused abou which ODS to be considered as a «reference»

3.4.4.3 Starting reading a Design

3.4.4.3.a Navigation

Although a HOOD documentation tends to produce “auto-sufficient documents for an object”, references to required objects lead often a reader to navigate in the different associated ODS.

hence it is always necessary to have a «map» of the object organisation: the system-configuration allows thus to define the context and the scope of that verification.

3.4.4.3.b "System_configuration"

A root object is defined by its interface with its environment. The latter should itself be defined as a partition of environment objects in the "system_configuration". Such environment objects may themselves be hierarchies. This allows:

- to factor the objects commons to several hierarchies as objects/environment hierarchies - common software, environmental software, preexisting to the system-to-design are thus modelled and integrated in the HOOD model-.
- represent the different subsystem of a subsystem at homogeneous abstraction levels, through different hierarchies. - a hierarchy representing a «view point» being selected, other related entities are defined as environment objects/hierarchies.

It is thus important to start the analysis and review of a design by the "system_configuration". It allows a first understanding of the partitioning of an architecture of system and of its environement.The system-to-design being itself described through several hierarchies.

3.4.4.3.c The Design Tree

For each hierarchy of objects or classes, the design tree represents the break down of the root object in child objects, on a variable number of decomposition levels.

The analysis of the tree structure gives thus a global view point on an architecture.
A design reviewer shall always have in mind the design tree in order to follow, its own navigation philosophy. We recommend to navigate «horizontally», level by level down to a level near the terminal objects.

At that time, it may be suitable to finish by «vertical navigation», since low level objects have very little to do one with another if they belong to distinct hierarchies.

Moreover it may be interesting to look globally at subsystems: in certain designs, subsystems are highly related and it may useful to shift on vertical navigation.

The analysis of the design tree may also highlight some problems with respect to the quality of the design. One may observe for example that some similar objects have been developed several times: they could have be «shifted» to an upper level as common objects or defined as environment objects of the current tree.

Also errors in understanding the HOOD method can be detected such as useless objects or decomposition levels.(décomposition of an object into a sole object or into only OP_CONTROL ones).

Finally examining the structure of a design tree may highlight a very strange partitioning, that needs for the least a well justified design decisions.

3.4.4.4 ODSs Readings

When reading an ODS, some simple rules may be applied :

- ODS analysis should be done tree by tree.
- the evaluation should start at level 0.
- an evaluation should first check the implementation of a parent specifications into child objects.
- an evaluation shall than trace requirement support child by child

It is also recommended to check the REQUIRED_INTERFACE since this allows also to detect inconsistencies in the use of the HOOD method, such as e.g.the call by a parent object of an operation provided by a child.

3.4.4.5 Redundancy Management

When reading a HOOD documentation, one may have some difficulties due some time to redundant parts between the descriptions appearing in the parent ODS and the ones appearing in child ODSs.
Such redundancy may rather add volume to the documentation to read, understand or to evaluate. One acceptable compromise may be to allow within a child ODS field to make references (instead of copy) to a parent ODS field thus avoiding duplicated textual descriptions.
However such referencing shall not be systematic: it should only be allowed for the parent information which is not distributed among several children. Moreover it should not be used between several level of decomposition since this would render the reading of a child at level4 very uncomfortable.
Knowing that the information contained in the fields of a parent ODS is generally distributed in some child ODS fields(such type of redundancy is unavoidable), it is necessary to check its consistency:

- If the text elaborated in the parent ODS fields is refined in the child ODS fields and is no more consistent/compatible with the parent ones, it has to be pointed and correct the information in the parent ODS.
- if the text elaborated in the parent ODS fields refers to the child ODS fields, this indicates in general that the documentation of the parent object has been produced after the one of the child ODSs: such a documentation should not be accepted since:
  - it indicates that the method has generally not be applied in the top-down way (may be even that the code was produced first!)
  - the documentation has been purely produced because it was requested, but it was for no help in the elaboration of the design. (and hence we do not need to spend any time in checking it)

3.4.4.6 Evaluation Process

The general evaluation process follows roughly the three following steps:

- check consistency of decomposition (are parent-child descriptions consistent?),
- check traceability (have all requirements been taken into account and where?),
- evaluate the design with respect to software engineering quality criteria (do we have a e "good design"?).

According to the background of the reader, the way these steps are performed may vary. A method consultant will primarily check for errors in understanding the method and with regard to software engineering quality criteria.

The customer will first check out requirements implementation and support and testability.

Finally quality control people will first check the consistency of parent-child descriptions.

But it is clear that a customer should also validate the quality of the design which is delivered to him. Based on the three steps described above, we suggest and recommend an approach like the following:

- Evaluate the design through "successive validation steps", i.e. by checking one after another the decompositions of parent objects into child ones, level after level. Although it is not always the case, at a given level, the abstraction «power» used is the same.
- Before going in the next level of decomposition and for the such prevalidated ODSs check the traceability analyses.
- Finally evaluate the decompositions through quality criteria (reuse, testability, Software Engineering quality, HOOD rules...)

All these activities may be done in a collaborative way by people having different backgrounds. But all must have some knowledge of the HOOD method (rather high for people evaluating design quality) as well as a good knowledge of the requirements and its environment (rather high for people checking traceability).

3.4.4.7 Managing Author-Reader Cycles

Author-readers cycles have always been a problem due to schedule constraints and delays. A HOOD document should be submitted to a reader with comments given backs within a given week. Otherwise the delay is too important and the process may hinder the design elaboration. The idea here is also that:

- when trying to apply strong author-reader cycles to all sub sections of an ODS DESCRIPTION field, one may hinder the creative process and responsibilities of the designer.
- when receiving ODS parts on which you, as a reader, almost fully agree, you may not even respond.

Such regulating rules tend to reduce dramatically the exchange rate of design documentation for author-reader cycles and rendering globally to a better process.

Moreover as experience becomes larger there are some particular sections of the ODS which are highly important for readers (namely the informal strategy and the operations descriptions).A

project should thus organize and plan the author-reader cycles so that the readers go prepared to give their best effort according to a planned delivery of these sections:

- either they receive continuous information from the writers and they may foresee when the real thing will be ready for review
- either they have to be ready to respond as soon as they receive the important sections.

3.4.5 TUTORING

Tutoring is supporting «on-line» a project team just trained in a new technology:

- by looking over their shoulders to check that the new technology is well used and applied
- by providing thus additional support and training at points where it is needed

Tutoring should be applied to HOOD projects, especially when the team is new or has very few HOOD practice.

When launching a HOOD project, it is mandatory to allocate a recognized HOOD tutor to the team: experience has shown that trying to correct errors late (e.g. at the time of the ADR) may cost very more and has fewer chance of success.

It thus important that the tutor is available right at the elaboration of the top-level design and that he injects back all its experience to the new team. Area of expertise where the tutor may jump in may vary:

- launching the project, configuring the HOOD toolsets, development environment
- defining the project approach
- author-reader cycles on the first informal strategies

etc...

4 METHOD SUPPORT AND EVOLUTION

4.1 THE HOOD USER GROUP

Early HOOD users have founded the HOOD User Group [4] which has as objectives exchanges of experience, data and method requirements repository, and evolution control of the HOOD method, thus:

- warranting the method stability,
- taking into account users needs,
- warranting method output and user design perenniality
- taking into account technological and standards evolution

The HUG means are the set-up of HOOD Working Groups and a technical committee (HOOD Technical Group) which validates and agrees on the HWG recommendations and technical documents such as:

- definition of textual and graphical formalism, code generation rules and Standard Interchange Format in the reference manual [1]
- definition of a Method User Manual[5] released in 1994.
- definition of method evolution (draft HOOD revision 4 integrating inheritance currently prototyped)

4.2 THE STANDARD INTERCHANGE FORMAT

The HOOD Reference Manual [1] defines a standard exchange format for the design description associated to HOOD design models and called SIF (Standard Interchange Format). This format, now supported by the HOOD toolsets benefits to HOOD users as

- a standardised ASCII exchange format for HOOD designs which is supported by HOOD toolsets.
- a mean of reuse and preserve the investment in producing a HOOD design: since HOOD descriptions will be able to be processed by current tools or by future tools
- a communication vehicle and exchange mean for the different teams in a large project. Teams may not have the same tools and still share their designs.
- a bridge towards a variety of other development tools (performance evaluation, traceability, analysis, documentation or reuse tools[6])

4.3 TOOLSETS

The tools supporting the method should enable graphical representation, textual edition, documentation edition and management, design checking and automatic production of code and test unit skeletons. There are now several toolsets available, which support more or less all these functions. Some tools also offer cross-reference generation, documentation configuration according to standards (ESA, COLUMBUS, DOD2167).
For a potential user, and a part from the user interface which is more or less going to standardise according to MOTIF standards, an important evaluation criteria is the support of the HOOD 3.1 and especially:

- generation of object descriptions in SIF format,
- import of object descriptions in SIF format This enables a user to change toolsets, without losing a database of designs produced with the old toolset. Also a user may work and develop with the toolset of his choice, and finally integrate and deliver on customer HOOD tool.
- Support of *System-Configuration* allowing easy reuse of existing designs and classes, and definition of the scope of the design checking space.

5 CONCLUSIONS

We have shown that the possibilities of the HOOD method allow to set up development strategies with improved support for the production, decomposition, refinement, test and reuse of models.

Today HOOD offers thus several advantages in terms of contribution to productivity and quality, especially when used with integrated tools such as CASE (Computer Aided Software Engineering) environments.

Productivity is improved thanks to a hierarchical decomposition approach, allowing the progressive development of objects by modelling, and their decomposition into software objects and abstract data types.

Although some HOOD users already push for new improvements (such as refinement of links between objects as new HOOD objects, or integration of inheritance), the HOOD 3.1 will be stable until the industrial spreading of Ada 9X and the publication of a standard for C++[9]. In the meantime HOOD extensions, prototyping and experimentation rely on the SIF.

The main extension work conducted so far is aiming at:

- defining methodological approaches for integrating different notations and methods, especially Object Oriented Languages such as C++ and Ada9X. A HOOD4 proposal was widely under discussion and adopted by the HUG.
- extending HOOD application towards *system design* where more formal representation of interfaces and behaviour for simulation and evaluation are strong requirements.

The HOOD development approach allows to combine both a design representation target towards flat type structure and object oriented class structure. Whereas several methods for identification of classes are based on analysis techniques that were mainly derived from the Entity-Relationship model extended with inheritance, the HOOD design approach leads naturally to the identification of class structures from the definition of logical interfaces and applicative abstract data types. The resulting structure reflects the top down design trade off partition of the software, rather than a bottom up approach derived from implementation data structures.

Taking into account both the natural client-server relationship between objects and classes, orthogonal to the composition relationships, this approach proves to be a powerful structuring tool. It appears today for us, as the only viable integration support for both modular and full object oriented approaches
Furthermore, by integrating both abstract data typing and process concepts HOOD3.1 is the software engineering tool of choice, supporting a smooth transition from classical development practice into full object oriented one.

6 BIBLIOGRAPHY

6.1 REFERENCES

[1]

HOOD Technical Group, B.DELATTE, M.HEITZ, JF MULLER editors, “HOOD Reference Manual”, *Prentice Hall and Masson, 1993.*

[2]

M.HEITZ / Cisi-Ingenierie, “Towards more formal developments through integration of behaviour expression notations and methods within HOOD developments” in *Proceedings of 5th International Conference on Software Engineering*, Toulouse, December 1992, Editions EC2

[3]

S.Mullender, “Distributed Systems”, ACM PRESS,ADDISON WESLEY 1991

[4]

“HOOD USERs GROUP” C/O SPACEBEL INFORMATIQUE, 111, rue Colonel BOURG, B-1140 BRUSSELS, BELGIUM Tel(32).2.730.46.50 fax (32) 2.726.85.13

[5]

HOOD Technical Group, B.DELATTE, M.HEITZ, JF MULLER editors, “HOOD User Manual”, HUG, C/O SPACEBEL INFORMATIQUE, 111, rue Colonel BOURG, B-1140 BRUSSELS, BELGIUM Tel(32).2.730.46.50 fax (32) 2.726.85.13

[6]

B.DELATTE,M.HEITZ,“Experimenting Reuse with HOOD and Ada” in *Proceedings of Ada- Aerospace Conference, BRUSSELS*, November 1993.

[7]

ISO/IEC JTC1/SC22 N1451 Ada9X Reference MANUAL, Version 4.0, September 1993

[8]

ISO/IEC JTC1/SC22 N1382 Draft Standard for EXTensions for Real-Time Ada version 3.0, october 1993

[9]

ANSI X3/J16 Draft proposal for the programming language C++, AT&T, June 1993

[ACM89]:

Davis: A Comparison of Techniques for the specification of external System Behavior, Communications of the ACM, Vol. 31 n9, 1989

AQS91]

"Ada Quality and Style ; Guidelines froProfessional Ada Programmers" , Version 02, Software Productivity Consortium, 1991

[ASX94]:

Schmidt D, ASX ('1994): An Object-Oriented Framework for Developping Distributed Applications, P*roceedings of the 6th USENIX C++ Conference, Cambridge, MA, April 1994*

[BERARD85]

Ed.Berard , "Object Oriented Design Handbook for Ada Software", EVB document1985

[BERRY88]:

G. Berry, Ph Couronne, G Gonthier: Synchronous Programming of Reactive Systems: an introduction to Esterel, INRIA Report 647 1988

[BOO 87]

G.Booch , "Software Componenet with Ada, Structures, Tools and Subsystems", The Benjamin/Cummings Publishing Company, 1987

[BOO86]

Booch Grady, Object-Oriented Development, IEEE Transactions on Software Engineering Vol SE-12, February 86

[BOO 83]

G.Booch , "Software Engineering with Ada", The Benjamin/Cummings Publishing Company, 1983

[CAR 91]

JR. Carter, "Concurrent Reusable Abstract Data Types", ACM Ada Letters Vol XI, Number 1, jan 1991

[CAR 90]

JR. Carter, "The Form of Reusable Ada Componenets for Concurrent Use", ACM Ada Letters Vol X, Number 1, jan 1990

[CCITT89]

CCITT (International Telecommunication Union), Instructions for SDL Users, Recommendation Z 100 - Annex D, Vol. X, Fasc. X.2, Geneva 1989

[Clark86]

E.M. Clarke, E.A. Emerson and A.P. Sistla, Automatic verification of finite state concurrent systems using temporal logic specifications, ACM Trans. Prog. Lang. Syst. 8 (1986) 244-263

[GIOVA89]

R. Di Giovanni, P.L. Iachini, HOOD and Z for the development of Complex Software Systems, Lecture Notes in Computer Science, n° 428, ' VDM'90, VDM and Z, FORMAL METHODS in SOFTWARE DEVELOPMENTS.

[GON 90]

D.W.Gonzalez, "Multi Tasking Software Components", ACM Ada Letters Vol X, Number 1, jan 1990

[Harel85]

D. Harel and A. Pnueli, On the development of reactive system, in: K.R. Apt, ED., Logics and Models of Concurrent Systems (Springer, New York, 1985) 477-498

[HEI91]

A.G. Heibig, Control Machines: a new model of parallelism for compositional specifications and their effective compilation, Theoretical Computer Science, Elsevier 1991 43-80

[HRM91]

HOOD Reference Manual (issue 3.1) - HOOD User Group -edited by MASSON and PRENTICE HALL available December1992.

[HUG91]

HOOD USER GROUP - HUG C/O SPACEBEL INFORMATIQUE, Chaussée de la Hulpe , B-1140 BRUSSELS, BELGIUM Tel(32).2.730.46.50 fax (32) 2.726.85.13

[HRT]

ESA/ESTEC, " Hard Real-Time Operating System Kernal Study",Task1 - Overview and Selection of Hard real Time Scheduling Model, ESTEC conttract n9198/90/NL/SF, also on the HOOD WEB site

[LECMP89]

V. Lecompte: Verification Automatique de Programmes Esterel, Thèse de doctorat de l'Université de Paris 7 - Juin 1989

[MHIS90]

M. Heitz, I. Sneed: HOOD-Esterel: A Methodology for Formal Design and Development of Real-Time Systems, Proceedings of EuroSpace Conference Barcelona december 1990.

[Mic 91]

M. Micouin, Objets et automates : un essai de mise en œuvre de Grafcet en conception HOOD, Conférence LIANA 1991 "Pratique et outils logiciels d'aide à la conception de systèmes d'information", Nantes

[Mull89]

S. Mullender, Distributed Systems, ACM PRESS, ADDISON WESLEY 1991

[MEYER90]

Meyer B,(1990)"*Object Oriented Software Construction"* in *ISBN 0-8053-0091*, Benjamin Cummings

[OMG91

OMG,(1991)*The Common Object request Broker : Architecture and Specification* , OMG doc.

[PALU90]

Vallette, Paludetto et Courvoisier, LAAS, Generation de code Ada en conception orientée Objects HOOD/Réseaux de Petri, Proceedings des Journées Internationales "Le Génie Logiciel et ses Applications", Toulouse, Décembre 90.

[Reisig85]

W. Reisig, Petri Nets: An Introduction (Springer, Berlin, 1985)

[SCHMIDT95]:

Schmidt D,Stephenson P(1995) Using Design Patterns to Evolve System Software from Unix to WIndows NT *C++ Report*, 1995 March 1995

[SOUROU95]:

Sourouille JL, H Lecoeuche(1995), Integrating State in an OO Concurrent Model, *Proceedings of TOOLS EUROPE95 Conference*, Prentice HAll

[STROUS91]

Strousoup B(1991) *The Annoted C++ Reference Manual*, Addison-Wesley Press

[VIEL89]

P. Vielcanet: HOOD Design Method and Control Command Techniques for the Development of Real-Time Software, Proceedings of ADA Conference, Washington, June

[VINO93]

Vinoski S(1993) Distributed Object Computing with CORBA, *C++ Report*, vol 5,

[VINO95]

Vinoski S, Schmidt D(1995) Comparing Alternative Client_Side Distributed Programming Techniques, *C++ Report*, 1995 May/June 1995 issues

[PFA85]

G.E. Pfaff "User Interface Management Systems" (Seeheim 1983), Springer-Verlag

6.2 HOOD BIBLIOGRAPHY

6.2.1 ARTICLES AND PAPERS PUBLISHED IN 1995

[MH95b] Achieving Reusable and Reliable Client-Server Code using HOOD automated code generation for ADA95 and C++ targets,M.HEITZ, Lecture Notes In Computer Science - SPRINGER VERLAG, Proceedings of the Ada-Eurospace Conference, Frankfurt, October 1995

[MH95a] Automated Client_Server Code Generation from HOOD4 Object Oriented Desigsn , M.HEITZ, Proceedings of the IFAC Workshop on Distributed Computer Control Systems Conference, Blagnac, September 1995

6.2.2 ARTICLES AND PAPERS PUBLISHED IN 1994

[MH94c] Integrating Modular, Object Oriented Programming and Application generator technologies in large Real Time and distributed Developments M.HEITZ, the Ada-Eurospace Conference, Copenhagen, September 1994,

[MH94b] EXPERIMENTING REUSE with HOOD and Ada, M.HEITZ,B.DELATTE, Proceedings of the Ada-Eurospace Conference, Brussels, November 1993

[MH94a] Analyse et Conception Orientée Objet avec HOOD, M. HEITZ, Journée Rencontres Industrielles ENSIMAG, Grenoble juin 1993.

6.2.3 ARTICLES AND PAPERS PUBLISHED IN 1993

[HRM93] HOOD Reference Manual 3.1, HOOD Technical Group, B.DELATTE, M.HEITZ,JF MULLER, Editions MASSON et PRENTICE HALL 1993.

[MH93b] EXPERIMENTING REUSE with HOOD and Ada, M.HEITZ,B.DELATTE, Proceedings of the Ada-Eurospace Conference, Brussels, November 1993

[MH93a] Analyse et Conception Orientée Objet avec HOOD, M. HEITZ, Proceedings de la conférence des Utilisateurs Informatique du CEA, Grenoble juin 1993.

[MHULL93] Introducing formal methods to HOOD, M.E.C. Hull & P.G. o'Donoghe, University of Ulster, Proceedings of the International Conference on Software Engineering Toulouse Nov. 1993.

[FM93] Ensembles de mesures applicables à la méthode HOOD, F.MAURICE, A.BENZEKRI, V.VALETTE, Proceedings of the International Conference on Software Engineering Toulouse nov 1993.

6.2.4 ARTICLES AND PAPERS PUBLISHED IN 1992

[ANDER92a] HOOD as a basis for OBJECT_ORIENTED PROGRAMMING J.A. ANDERSON (SYNETICS) Proceedings of the Ada-Eurospace Conference, Vienna, No-

vember 1992

[ANDER92b] INCORPORATING MANAGEMENT INDICATORS INTO HOOD J.A. ANDERSON(SYNETICS) Proceedings of the Ada-Eurospace Conference, Vienna, November 1992

[ANDR92] Reverse Engineering from Ada to HOOD P.ANDRIEU, C.ALVAREZ, JA. ESTEBAN (SEMA group) Proceedings of the Ada-Eurospace Conference, Vienna, November 1992

[Chand92] Risk Reduction on Huygens Programme through Prototyping in the early Life Cycle Phases S.CHANDLER, P ADVIES, J HARBONNE (LOGICA) Proceedings of the Ada-Eurospace Conference, Vienna, November 1992

[CHIA92] Ada and HOOD for the development of the COLUMBUS Operating System Kernel, R.CHIAVERINI, S.TRIVELLINI, G.DELLARTE, S.SABINA (INTECS) Proceedings of the Ada-Eurospace Conference, Vienna, November 1992

[DEM92] An Xview application and the HOOD-Ada lifecycle B.DEMEUSE, O.LANGHENDRIES,G.PAQUET (SPACEBEL) Proceedings of the Ada-Eurospace Conference, Vienna, November 1992

[DENSK92] CASE and HOOD-Ada in the development of embedded systems U.DENSKAT, D.COATES,D.WEBER (DORNIER) Proceedings of the Ada-Eurospace Conference, Vienna, November 1992

[GAJET92] A HOOD-Ada Application for an Object Oriented Simulation Library for Onboard Space System Payload Simulation M.GAJETTI,W.RAVETTO (Alenia Spazio) Proceedings of the Ada-Eurospace Conference, Vienna, November 1992

[LAC92] Use of the HOOD Metgod in the HERMES Onboard Software P.LACAN,P.COLANGELI (Aerospatiale) Proceedings of the Ada-Eurospace Conference, Vienna, November 1992

[MAR92] The VICOS Software Development U.MARON (ERNO) Proceedings of the Ada-Eurospace Conference, Vienna, November 1992

[MARRD92] The Use of Ada and HOOD in the Design of an Autonomous Spacecraft Data Management System , L.MARRADI, F.CICERI,D.MASOTTI (LABEN) Proceedings of the Ada-Eurospace Conference, Vienna, November 1992

[MH92a] Specification et Conception en technologie Objet, l'approche HOOD, M. HEITZ, Conférence DECUS France, Avril 1992.

[MH92b] Towards more formal developments using HOOD, M. HEITZ, (CISI-INGENIERIE) Proceedings of the International Conference on Software Engineering Toulouse dec 1992.

[NAND92] Real Time On board Software for SILEX Space Communication System, P.NANDE (Matra Marconi Space) Proceedings of the Ada-Eurospace Conference, Vienna, November 1992

[PAPAIX92] HOOD & ADA Quality Engineering in french Aerospace Industry, M. PAPAIX,M. HEITZ, (CNES) Proceedings of the Ada-Eurospace Conference, Vienna, November 1992, EUROSPACE , 16bis avenue Bosquet 75007 PARIS

[POU92] Customizable MODULAR SOFTWARE, J.POUDRET, P GUEDOU, T GOHON (SEXTANT), Proceedings of the Ada-Eurospace Conference, Vienna, November 1992

[ROB92] HIERARCHICAL OBJECT ORIENTED DESIGN, PJ ROBINSON, Prentice

HALL

[RUN92] A control system for the X_SAR Precision Processor H. Runge, P. Secchi (DLR), A.Paganone (Intecs Sistemi), Proceedings of the Ada-Eurospace Conference, Vienna, November 1992

6.2.5 ARTICLES AND PAPERS PUBLISHED IN 1991

[And 91] Relations entre l'environnement HOOD et les autres composants de l'atelier CONCERTO, P. Andrieux, E. André, Conférence LIANA 1991 "Pratique et outils logiciels d'aide à la conception de systèmes d'information", Nantes

[Ander91] Object Oriented and Ada lifecycle reuse, J. Anderson, 2ème conférence EUROSPACE "Ada in Aerospace", Rome, Novembre 1991

[Bar 91] Eléments de la méthode HOOD Pour une composition ascendante d'objets, M. Bari, C. Rolland, Conférence LIANA 1991 "Pratique et outils logiciels d'aide à la conception de systèmes d'information", Nantes

[BD91a] La démarche HOOD au CNES, B. Delatte, Conférence LIANA 1991 "Pratique et outils logiciels d'aide à la conception de systèmes d'information", Nantes

[BD91b] Experimenting Reuse in French Aerospace Industry, B. Delatte, ?M. Heitz, 2ème conférence EUROSPACE "Ada in Aerospace", Rome, Novembre 1991

[Bea 91] Practical application of the rate monotonic scheduling theory with Ada, J. Beaufays, Ph. Bocquillon, Th. Dubois, E. Germeyn, H. Gilman, E. Snyers, 2ème conférence EUROSPACE "Ada in Aerospace", Rome, Novembre 1991

[Bir 91] EASAMS'Ariane5 on-board software experience, S.A. Birnie, 2ème conférence EUROSPACE "Ada in Aerospace", Rome, Novembre 1991

[Car 91] Unifying the concepts of HOOD with Object Oriented Programming, A.R. Carmichael, 2ème conférence EUROSPACE "Ada in Aerospace", Rome, Novembre 1991

[Deb 91] Description d'un générateur Ada d'objets HyperHood, Debœuf, Gatineau, Conférence LIANA 1991 "Pratique et outils logiciels d'aide à la conception de systèmes d'information", Nantes

[Do 91] A new approach to HOOD : A link between the design and the implementation, T.N. Do, 2ème conférence EUROSPACE "Ada in Aerospace", Rome, Novembre 1991

[Dona 91] Experience in using an object oriented method and Ada for the development of a Flight Management System, J.DONAGHY,(EUROCONTROL), Object Oriented Software Engineering Conference, Campus THOMSON, PARIS, sept 1991

[Habri 91] la méthode HOOD et quelques autres, H.HABRIAS, Conférence LIANA 1991 "Pratique et outils logiciels d'aide à la conception de systèmes d'information", Nantes

[Kou 91] Implantation de HOOD 3.0 dans un outil de génie logiciel existant, ?A. Kouri, P. Gendre, Conférence LIANA 1991 "Pratique et outils logiciels d'aide à la conception de systèmes d'information", Nantes

[Lai91] M.LAI, Conception Orientée Objet - la méthode HOOD?Editions A.COL-

IN.1991

[MH91a] HOOD3.1,the Adult Age, B.DELATTE/CNES, JF MULLER/MATRA-MARCONI-SPACE, M. HEITZ/CISI INGENIERIE, Proceedings of the Ada-Eurospace Conference, Roma, November 1991

[MH91b] Structuration des Classes, Objets, Types et Données dans HOOD, M. HEITZ, CISI INGENIERIE, Numero Spécial AFCET sur les Objets juin 1991.

[MH91c] Structuring Classes, Objects, Types and Data within HOOD, M. HEITZ, CISI INGENIERIE, Special Issue AFCET journal on objects june 1991.

[MH91d] Développement d'applications réparties avec ADA et HOOD, ?M. HEITZ, CISI INGENIERIE, Conférence ADA-FRANCE Nov 1991,Paris.

[Mic 91] Objets et automates : un essai de mise en œuvre de Grafcet en conception HOOD, M. Micouin, Conférence LIANA 1991 "Pratique et outils logiciels d'aide à la conception de systèmes d'information", Nantes

[Mon 91] The process of developping Ada software for large scale real-time and embedded systems, with distributed and redundant architecture, Ph. Monseu, 2ème conférence EUROSPACE "Ada in Aerospace", Rome, Novembre 1991

[Mul 91] Eléments de comparaison entre la méthode HOOD et les langage par objets, N. Cam, J.F. Muller, Conférence LIANA 1991 "Pratique et outils logiciels d'aide à la conception de systèmes d'information", Nantes

[Ponc91] De HOOD à HOOK : une méthode de conception uniforme pour ADA et les Bases de Données Orientées Objet, P. Poncelet, M. Teisseire, A. Cavarero, S. Miranda, Rapport K-I2S/HOOK, Janvier 1991

[Sch 91] Using the HOOD virtual node concept for the COLUMBUS DMS Software architectural design, F. Feck, B. Labreuille, J.F. Muller, H. Schindler, 2ème conférence EUROSPACE "Ada in Aerospace", Rome, Novembre 1991

[Seh 91] Prototyping for the basic management computer in helicopter project 'Tiger',H. Sehmer, 2ème conférence EUROSPACE "Ada in Aerospace", Rome, Novembre 1991

[Ser 91] Prototyping the CPF Hermes software with Ada, R. Serrao, M. Manigrasso, 2ème conférence EUROSPACE "Ada in Aerospace", Rome, Novembre 1991

[Tem 91] Some critical comments on HOOD, T. Tempelmeier, 2ème conférence EUROSPACE EUROSPACE "Ada in Aerospace", Rome, Novembre 1991

[Tra 91] Mise en œuvre de la méthode HOOD pour la simulation discrète de systèmes distribués avec le langage Ada, M.K. Traore, Conférence LIANA 1991 "Pratique et outils logiciels d'aide à la conception de systèmes d'information", Nantes

[Wil 91] Building a system for screening ERS-1 satellite data using the HOOD method and Ada, M. Eineder, H.J. Wilhelms, P; Coppola, A. Paganone, 2ème conférence EUROSPACE "Ada in Aerospace", Rome, Novembre 1991

6.2.6 ARTICLES AND PAPERS PUBLISHED IN 1990

[Auton90] Use of HOOD in the Design of a Multiprocessor Real Time System, I. Auton, Thorn Emi, Proceedings of Hood conference, April 1990, Manchester Airport

[BARCE90] MSLIB_Ada : a library for spatial mechanics computing, JC BERGES (CNES°, C.COMPE, L.MALLET (SIMULOG), F.DUCRET (AEROSPATIALE), Proceedings of the Ada-Eurospace Conference, Barcelona, december 1990

[BARCE90] Prototyping in Ada : experience in developing the HERMES on-board mission software mockup, A;BARCELLONA, A.SANTARELLI, Proceedings of the Ada-Eurospace Conference, Barcelona, december 1990

[BING90] Ada realization for the process computers of the DOEL1&é nuclera power plants, M.BINGEN, J.BEAUFAYS, Proceedings of the Ada-Eurospace Conference, Barcelona, december 1990

[BL90] HOOD , A Design Method for Space Industry JF Muller, B.Labreuille/ MATRA-ESPACE, Proceedings of American Institute of Aeronautics and Astronautics Conference Sept 1990, PASEDENA.

[GEN90] HOOD version 3.0 : l'âge de raison, P. Gendre et H. Bitteur, Génie Logiciel et Système Expert, Vol. n° 19, pp 44-48, juin 1990

[GIOV90] On the translation of HOOD Nest into Ada, R.Di Giovanni (PRISMA), Proceedings of ADA-Europe, Dublin 1990, Cambridge University Press, ADA Companion series

[IS 90] Integration of H_OOD ,Estérel and Ada for the development of safe and reliable Real Time Systems, I. SNEED, M. HEITZ, CISI INGENIERIE, Proceedings of the Ada-Eurospace Conference, Barcelona, december 1990

[JOCT90] La méthode HOOD, compte-rendu du groupe de travail AFCET, H JOCTEUR, AFCET -INTERFACES n°93, Juillet 90

[LAFEU90] Contribution of Ada to an improved Software Development Approach, Y.LAFEUILLADE, F.COURJARET (MATRA-MS2I) Proceedings of the Ada-Eurospace Conference, Barcelona, december 1990

[Lai 90] An overview of several French Navy projects using HOOD and Ada, M. Lai, Proceedings of the Ada-Europe conference 1990 Dublin, ADA Companion Series, Ada Cambridge University Press 1989

[LARR90] A Baseline for a HOOD method Assistant, JR Larre, N.Alfaro, JJ.GAlan, J.Garbaosa, GMV:Spain Proceedings of the Ada-Eurospace Conference, Barcelona, december 1990

[Mell90] A Pragmatic Application of OOD to the Development of Ada Using the Teamwork Toolset, P. Mellor, Proceedings of Hood conference, April 1990, Manchester Airport

[MH90a] La méthode de conception HOOD dans le cycle de développement et ses outils, M.Heitz, Proceedings des Journées Internationales de l'Informatique et de l'Automatisme, JIIA 90

[MH90b] Integration of HOOD in the Lifecycle, M. Heitz, Proceedings of Hood conference, April 1990, Manchester Airport

[MU90] HOOD, A Method to Support Real Time and Embedded Systems Design ? JF Muller/MATRA-ESPACE, Jochen DERISSEN/GEI, M.Heitz and I.SNEED/ Cisi-Ingenierie,Proceedings of Journees du Genie Logiciel Toulouse, Décembre 1990, Editions EC2

[Neil90] Use of Yourdon for Requirements Analysis in combination with HOOD for Architectural Design, J. van Neil, Proceedings of Hood conference, April 1990,

Manchester Airport

[NIEL90] Extension of Communication Facilities in Ada, HJ Goedman & J Van NIEL (HCS) Proceedings of HOOD conference, April 1990, Manchester Airport

[PALU90] Generation de code Ada en conception orientée Objects HOOD/Réseaux de Petri, VALLETTE, PALUDETTO et COURVOISIER, LAAS, Proceedings des Journées Internationales "Le Génie Logiciel et ses Applications", Toulouse, Décembre 90.

6.2.7 ARTICLES AND PAPERS PUBLISHED IN 1989, 1988 ET 1987

[AUX89] Promethée : Designing a Process Control System, G.Auxiette/ TOTAL, ?JF CABADI, P.Rehbinder/Cisi-Ingénierie, Proceedings of ADA-EUROPE Conference 89, Madrid

[BL 88] Design and Development of Distributed Software using Hierarchical Object Oriented Design and ADA, M. HEITZ, B. LABREUILLE, CISI INGENIERIE, Proceedings Ada Europe Conference München, May 88.

[CAR89] Integrated Support for Requirements Analysis, HOOD Design and Implementation, A.R. Carmichael, UNICOM, HOOD Tutorials and Demonstrations", Londres, 28-29 novembre 1989

[GIOVA89] HOOD and Z for the development of Complex Software Systems, R. Di Giovanni, P.L. Iachini, Lecture Notes in Computer Science, n° 428, ' VDM'90, VDM and Z, FORMAL METHODS in SOFTWARE DEVELOPMENTS.

[HEI 89-1] Hierarchical Object Oriented Design for Ada development of large technical and realtime software, M. HEITZ, technical Note CISI INGENIERIE, 3 pages, 21/06/89.

[HEI 89-2] Hierarchical Object Oriented Design - Une méthode de conception orientée Ada pour le développement de logiciels techniques et temps réels, M.HEITZ, Note technique CISI INGENIERIE, 3 pages, 08/12/89.

[HEI 89] Hierarchical Object Oriented Design - A Summary, M. HEITZ, CISI INGENIERIE, a 10 page summary, 13/11/89.

[HRM 89] H_OOD Reference Manual, V3.0, produced by HOOD Working Group, edited by ESA Noordwijk, The Netherlands, 09/89

[HUM 89] H_OOD User Manual, V3.0, produced by HOOD Working Group, edited by ESA Noordwijk, The Netherlands, 12/89

[IS 89] An Extended Algebraic Approach for Methods Formalisation, Application to the HOOD Method, Deuxièmes Journées Internationales "Le Génie Logiciel et ses Applications", I. SNEED, M. HEITZ, B. RAYNAUD, Proceedings of Journées du Génie Logiciel Toulouse, 4-8 Décembre 1989

[VIEL 89] H_OOD Design method and control/command techniques for the development of realtime software, P. VIELCANET, CISI INGENIERIE, Proceedings of ADA Conference, Washington, 06/89.

[HEI 88] Hierarchical Object Oriented Design for Ada, La Lettre ADA n° 8, M.HEITZ, CISI INGENIERIE, 2 pages 10/88.

[VAL 88] Réseaux de Pétri pour la conception de logiciels temps réels, VALLETTE,

PALUDETTO, du LAAS, et LABREUILLE, CISI INGENIERIE, Proceedings des Premières Journées Internationales "Le Génie Logiciel et ses Applications", Toulouse, Décembre 88.

[HEI 87] H_OOD, Une Méthode de Conception Hierarchisée Orientée Objects pour les développements des logiciels techniques et Temps réels, M.HEITZ, Proceedings Journées ADA-AFCET, 12/87.

A APPENDIXES

A1 MORE ON THE EMS EXAMPLE

A1.1 EMS REQUIREMENTS

A1.1.1 PRESENTATION OF THE EMS SYSTEM

The EMS is a software to monitor a motor engine variables, to display them on a bargraph and trigger an alarm if associoated values are out of range. A detailed requirement of the EMS is given hereafter

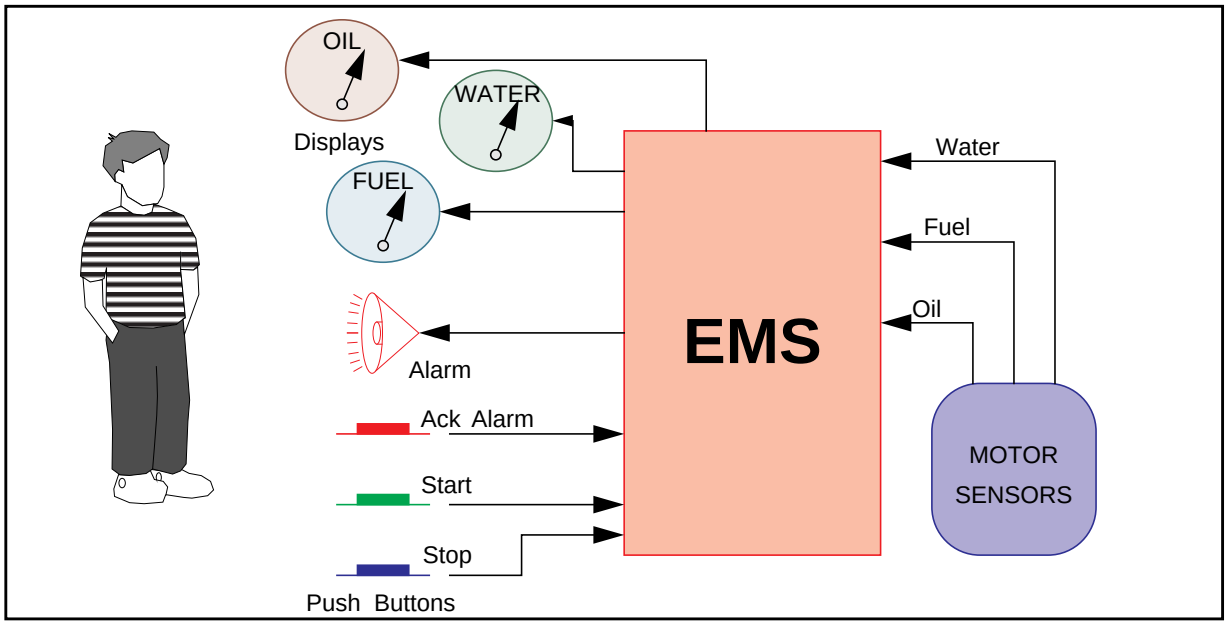


Figure 158 - The EMS System

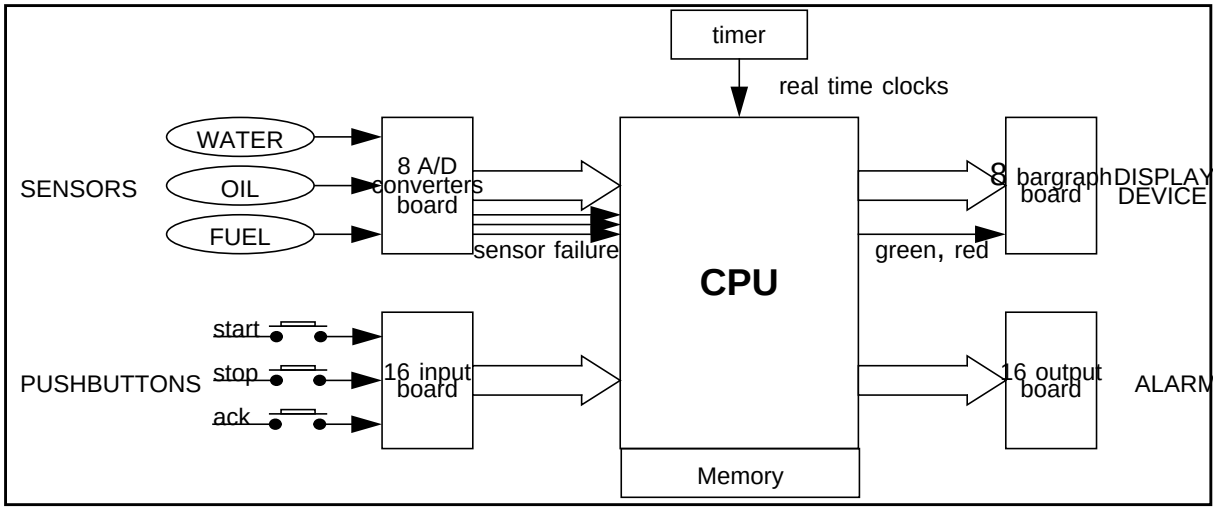


Figure 159 - The EMS Hardware Block Diagram

A1.1.2 CLIENT REQUIREMENTS

- The EMS software is running on the CPU, using an Input/Output and a Timers drivers.
- The values of water temperature, oil pressure and fuel level are displayed in green light on three different bargraphs, each second. In case of sensor failure, the corresponding bargraph flashes in red light and the alarm is started. In case of a value is out of range, the bargraph displays the value in red light and triggers the alarm.
- The EMS can be started and stopped independently of the car engine, by using start and stop pushbuttons.
- The alarm of the EMS may be switched off by the ack push button.
- More sensors might be handled.
- The EMS must be reliable.
- The EMS system might be connected to a general vehicle control system.

A1.1.3 SOFTWARE ENVIRONMENT

- The Timers_Driver object manages a set of timers which send cyclic interruptions at a specified address. Each timer is first created with given frequency and address. A timer then may be started or stopped at any moment. It may also be deleted from the list of available timers.
- The Input_Output_Driver object provides means to exchange information with hardware devices. It allows to put or to get information of these devices (excepted the push-buttons). Malfunction from a hardware device is returned as an exception of the put or get operation. Pushing a push-button generates an interruption.

A1.2 OBJECT AND OPERATION IDENTIFICATION THROUGH TEXTUAL ANALYSIS TECHNIQUES

A1.2.1 IDENTIFICATION OF NOUNS :

On IT reception from the Start push button, the **EMS** is started if it was not.
Then the **sensors** of the **EMS** are sampled every 1/10 second by a signal from the **timer**.
Every second, the **mean values of oil pressure**, **fuel level** and **water temperature**, are acquired from **sensors** and then displayed on the **appropriate bargraphs**. If a **mean value** is out of range, the **alarm** is switched on and the corresponding **bargraph** is displayed in **red light**. If there is a **sensor failure**, the **alarm** is switched on and the corresponding **bargraph** flashes in **red light**. In the other cases, the **bargraphs** are displayed in **green light**.
On IT reception from the Ack push button, the **alarm** is acknowledged if it was not.
On IT reception from the Stop push button, the **EMS** is stopped if it was not : the **alarm**, the **timer** and the **sensors** are stopped and the **bargraphs** are switched off.

NOUNS	IDENTIFIERS	OBJECT	DATA	COMMENTS
EMS	EMS	X		System to be designed
sensors	Sensors	X		3 sensors
timer	Timers_Driver	X		environment
mean value of oil pressure	value		X	
mean value of water temperature	value		X	
mean value of fuel level	value		X	
appropriate bargraphs	Bargraphs	X		3 bargraphs
mean value	value		X	
range	limited_value		X	system parameters
alarm	Alarm	X		
red light	colour		X	
sensor failure	sensor_failure		X	
green light	colour		X	

A1.2.2 IDENTIFICATION OF VERBS (❸ CONTINUE)

On IT reception from the Start push button, the EMS **is started** if it was not.
Then the sensors of the EMS **are sampled** every 1/10 second by a signal from the timer.
Every second, the mean values of oil pressure, fuel level and water temperature, **are acquired** from sensors and then **displayed** on the appropriate bargraphs. If a mean value **is out of range**, the alarm **is switched on** and the corresponding bargraph **is displayed** in red light. If **there is a sensor failure**, the alarm **is switched on** and the corresponding bargraph **flashes** in red light. In the other cases, the bargraphs **are displayed** in green light.
On IT reception from the Ack push button, the alarm **is acknowledged** if it was not.
On IT reception from the Stop push button, the EMS **is stopped** if it was not : the alarm, the timer and the sensors **are stopped** and the bargraphs **are switched off**.
Grouping Operations and Objects (❸ continue)

VERBS	Server OBJECT	IDENTIFIERS	COMMENTS (execution request, para
started	EMS	start	on IT from Start button
sampled	Sensors	sample	every 1/10 second by a signal from timer
∑ 8 following actions	?	monitor	every second
acquired mean value	Sensors	acquire	
displayed_mean_val.	Bargraphs	display	
is out of range	? or Filter	is_value_outof_range	
switched on	Alarm	switch_on	
displayed	Bargraphs	set_colour	in red light
there is sensor failure	Sensor	is_there_sensor_fail.	
flashes	Bargraphs	flash	in red light
displayed	Bargraphs	set_colour	in green light
acknowledged	EMS/Alarm	acknowledge	on IT from Ack button
stopped =	EMS	stop	on IT from Stop button
stopped	Alarm	stop	
stopped	Timers_Driver	stop	environment
stopped	Sensors	stop	
switched off	Bargraphs	switch_off	

A1.3 OTHER ODS OF EMS SYSTEM

A1.3.1 OBJECT TIMERS_DRIVER IS ENVIRONMENT ACTIVE

DESCRIPTION
--| The Timers_Driver manages a set of timers which send cyclic interruptions at a specified address. A timer is created with a given frequency and may be started (re-started) or stopped at any moment. A timer then may be deleted from the list of available timers.|--
IMPLEMENTATION_CONSTRAINTS
--| 255 timers may be consecutively used at a same time.|--
PROVIDED_INTERFACE
TYPES
T_TIMER; --| This type defines a timer.|--
T_IT_ADDRESS is SYSTEM.ADDRESS; --| This type defines the address of an IT.|--
CONSTANTS
NONE
OPERATION_SETS
NONE
OPERATIONS
Create(TIMER : out T_TIMER; -- new timer
FREQUENCY : in POSITIVE; -- cyclic frequency of the timer in Hz
IT_ADDRESS : in ADDRESS -- reception address of the sent IT);
--| The Create operation allows to create a new timer, which can send IT at the address of reception and at the specified frequency. The timer is not started. The created timer is returned.|--
Start (TIMER : in T_TIMER -- timer to be started);
--| The Start operation allows to start the given timer.|--
Stop (TIMER : in T_TIMER -- timer to be stopped);
--| The Stop operation allows to stop the given timer.|--
Delete (TIMER : in T_TIMER -- timer to be deleted);
--| The Delete operation allows to suppress the given timer of the list of available timers. This operation stops the timer (if it is running) before to delete it.|--
EXCEPTIONS
NONE
OBJECT_CONTROL_STRUCTURE
DESCRIPTION
--| Start, Stop and Delete operations may arrive at any moment, as soon as the timer exists. A Start operation is of no effect if the timer is still running or is not created. A Stop operation is of no effect if the timer is still stopped or is not created. A Delete operation is of no effect if the timer is not created.|--
CONSTRAINED_OPERATIONS
Start CONSTRAINED_BY ASER;
Stop CONSTRAINED_BY ASER;
Delete CONSTRAINED_BY ASER;
END_OBJECT *Timers_Driver*

A1.3.2 OBJECT SENSORS IS ACTIVE

DESCRIPTION
--| This object samples oil pressure, water temperature and fuel level at 10Hz and stores the read values of the three sensors at any moment. It may provide the mean of stored values of a sensor.
It also provides all means to start and stop the hardware sensors.|--
IMPLEMENTATION_CONSTRAINTS
--| Buffering capacity, for each sensor, is of ten values.|--
PROVIDED_INTERFACE
TYPES

```
    T_SENSOR is(OIL_PRESSURE_SENSOR,
                WATER_TEMPERATURE_SENSOR,
                FUEL_LEVEL_SENSOR);
    --| This type defines the types of sensors which are taken into account in this object. |--
OPERATIONS
    Start;
    --| This operation initializes the hardware sensors.|--
    Sample;
    --| This operation is in charge of the Sensors sampling. It gets the current values of each hardware sensor and stores them
    into an internal database.|--
    Acquire (SENSOR : in T_SENSOR) return POSITIVE;
    --| This operation returns the mean of the ten last stored values of a sensor. It  returns a SENSOR_FAILURE exception in
    case of  hardware sensor problem.|--
    Stop;
    --| This operation stops the hardware sensors.|--
EXCEPTIONS
    SENSOR_FAILURE RAISED_BY Acquire;
OBJECT_CONTROL_STRUCTURE
DESCRIPTION
--|  The Sensors object accepts to sample as soon as it has been initialized. The Sample operation is not significant before a
    Start or after a Stop.|--
CONSTRAINED_OPERATIONS
Sample CONSTRAINED_BY ASER_by_IT Timer_10Hz;
REQUIRED_INTERFACE
OBJECT Input_Output_Driver;
    TYPES
        T_DEVICE;T_DEVICE_STATUS;T_DESCRIPTOR;
    CONSTANTS
        MAX_SENSORS_NUMBER;
    OPERATIONS
        Get;      Put;
    EXCEPTIONS
        MALFUNCTION;
OBJECT Timers_Driver;
    TYPES
        T_TIMER;
    OPERATIONS
        Create;  Start;   Delete;
DATAFLOWS
    data_in <= Input_Output_Driver;
    initial_frequency => Timers_Driver;
EXCEPTION_FLOWS
    malfunction <= Input_Output_Driver;
INTERNALS
OBJECTS
    NONE
TYPES
    T_SENSOR_STATUS is(INITIALIZED, STOPPED);
    T_HARDWARE_SENSOR_STATUS is(FAILED, OK);
    --| TBD |--
CONSTANTS
    IT_10HZ_ADDRESS : constant := Sample'ADDRESS;
    SAMPLING_FREQUENCY : constant :=10;
    --| TBD |--
OPERATIONS
    Store_Value(SENSOR : in T_SENSOR;
        VALUE : in POSITIVE;
        STATUS : in T_HARDWARE_SENSOR_STATUS);
    Get_Last_Value(SENSOR : in T_SENSOR;
        VALUE : out POSITIVE;
        STATUS : out T_HARDWARE_SENSOR_STATUS);
    --| TBD |--
```

EXCEPTIONS
--| TBD |--
DATA
SAMPLING_TIMER : Timers_Driver.T_TIMER;
WATER_SENSOR,
OIL_SENSOR,
FUEL_SENSOR : Input_Output_Driver.T_DEVICE (SENSOR);
SENSOR_STATUS : T_SENSOR_STATUS := STOPPED;
--| TBD |--
OBJECT_CONTROL_STRUCTURE
PSEUDO_CODE
--| TBD |--
CODE
--| TBD |--
OPERATION_CONTROL_STRUCTURES
OPERATION Start
DESCRIPTION
--| This operation initializes the hardware sensors and then creates and starts a timer for it triggers the sampling every 10 seconds. The sensors status are set to INITIALIZED.|--
USED_OPERATIONS
Timers_Driver.Create;
Timers_Driver.Start;
Input_Output_Driver.Put;
PROPAGATED_EXCEPTIONS
NONE
HANDLED_EXCEPTIONS
Input_Output_Driver.MALFUNCTION;
PSEUDO_CODE
--| TBD |--
CODE
--| TBD |--
END_OPERATION Start
OPERATION Sample
DESCRIPTION
--| This operation is in charge of the Sensors sampling. It gets the current values of each hardware sensor and stores them into an internal database.|--
USED_OPERATIONS
Input_Output_Driver.Get;
Store_Value;
PROPAGATED_EXCEPTIONS
NONE
HANDLED_EXCEPTIONS
Input_Output_Driver.MALFUNCTION;
PSEUDO_CODE
--| TBD |--
CODE
--| TBD |--
END_OPERATION Sample
OPERATION Acquire(SENSOR : in T_SENSOR) return POSITIVE
DESCRIPTION
--| This operation gets the ten last stored values of a sensor, calculates and returns their mean. In case of the corresponding hardware sensor is failed for the ten stored values, this operation returns a sensor_failure exception. In the other cases, a momentaneous hardware sensor failure is ignored and the mean value is calculated on the correct values.|--
USED_OPERATIONS
Get_Last_Value;
PROPAGATED_EXCEPTIONS
SENSOR_FAILURE;
HANDLED_EXCEPTIONS
NONE
PSEUDO_CODE
--| TBD |--
CODE

```
    --| TBD |--
END_OPERATION Acquire
OPERATION Stop
DESCRIPTION
--|  This operation stops the hardware sensors and resets the sampling timer. The sensors status is set to STOPPED.|--
USED_OPERATIONS
    Timers_Driver.Delete;
    Input_Output_Driver.Put;
PROPAGATED_EXCEPTIONS
    NONE
HANDLED_EXCEPTIONS
    Input_Output_Driver.MALFUNCTION;
PSEUDO_CODE
    --| TBD |--
CODE
    --| TBD |--
END_OPERATION Stop
OPERATION Store_Value (SENSOR : in T_SENSOR;
                       VALUE : in POSITIVE;
                       STATUS : in T_HARDWARE_SENSOR_STATUS)
DESCRIPTION
    --| TBD |--
USED_OPERATIONS
    --| TBD |--
PROPAGATED_EXCEPTIONS
    --| TBD |--
HANDLED_EXCEPTIONS
    --| TBD |--
PSEUDO_CODE
    --| TBD |--
CODE
END_OPERATION Store_VAlue
OPERATION Get_Last_Value (SENSOR : in T_SENSOR;
                          VALUE : out POSITIVE;
                          STATUS : out T_HARDWARE_SENSOR_STATUS)
DESCRIPTION
    --| TBD |--
USED_OPERATIONS
    --| TBD |--
PROPAGATED_EXCEPTIONS
    --| TBD |--
HANDLED_EXCEPTIONS
    --| TBD |--
PSEUDO_CODE
    --| TBD |--
CODE
END_OPERATION Get_Last_Value
END_OBJECT Sensors
```

A1.3.3 OBJECT BARGRAPHS IS PASSIVE

```
DESCRIPTION
--|  The bargraphs object allow to display values, in red or green with or without flashing, on appropriate display devices.
    It also provides all means to start and stop the hardware display_device.|--
IMPLEMENTATION_CONSTRAINTS
--|  NONE|--
PROVIDED_INTERFACE
TYPES
```

```
    T_BARGRAPH is(OIL_PRESSURE_BARGRAPH,
                  WATER_TEMPERATURE_BARGRAPH,
                  FUEL_LEVEL_BARGRAPH);
    --| Defines the type of bargraphs which are taken into account in this object.|--
    T_COLOUR is(RED, GREEN);
    --| Enumerated type giving the different displaying colours.|--
    T_FLASHING is(ON, OFF);
    --| Status of the displaying : Flashing or not.|--
    T_PERCENTAGE isRANGE 0..100;
    --| Type of the values which are displayed on the bargraphs.|--
OPERATIONS
    Init;
    --| This operation initializes the hardware bargraphs. Bargraphs are switched on in green colour without flashing and with
    a null value.|--
    Display(BARGRAPH : in T_BARGRAPH;
            VALUE : in T_PERCENTAGE);
    --| This operation displays the input percentage value in the corresponding hardware bargraph. The colour and the flashing
    are not modified.|--
    Set_Colour(BARGRAPH : in T_BARGRAPH;
       COLOUR : in T_COLOUR);
    --| This operation sets the corresponding hardware bargraph in the specified colour. The displayed value and the flashing
    are not modified.|--
    Flash(BARGRAPH : in T_BARGRAPH;
       STATUS : in T_FLASHING);
    --| This operation allows to flash the corresponding hardware bargraph. The colour and the displayed value are not modi-
    fied.|--
    Switch_Off;
    --| This operation switches the hardware bargraphs off.|--
REQUIRED_INTERFACE
OBJECT Input_Output_Driver;
    TYPES
       T_DEVICE;
       T_DEVICE_STATUS;
       T_BARGRAPH_COLOUR;
       T_BARGRAPH_FLASH_STATUS;
       T_DESCRIPTOR;
    CONSTANTS
       MAX_BARGRAPHS_NUMBER;
    OPERATIONS
       Put;
    EXCEPTIONS
       MALFUNCTION;
DATAFLOWS
    data_out => Input_Output_Driver;
EXCEPTION_FLOWS
    malfunction <= Input_Output_Driver;
INTERNALS
OBJECTS
    NONE
TYPES
    --| TBD |--
CONSTANTS
    --| TBD |--
OPERATIONS
    --| TBD |--
DATA
    WATER_BARGRAPH,
    OIL_BARGRAPH,
    FUEL_BARGRAPH : Input_Output_Driver.T_DEVICE (BARGRAPH);
    --| TBD |--
OPERATION_CONTROL_STRUCTURES
OPERATION Init
```

```
DESCRIPTION
--| This operation initializes the hardware bargraphs. Bargraphs are switched on in green colour without flashing and with a
    null value.|--
USED_OPERATIONS
    Input_Output_Driver.Put;
PROPAGATED_EXCEPTIONS
    NONE
HANDLED_EXCEPTIONS
    Input_Output_Driver.MALFUNCTION;
PSEUDO_CODE
    --| TBD |--
CODE
    --| TBD |--
END_OPERATION Init
OPERATION Display(BARGRAPH : in T_BARGRAPH; VALUE : in T_PERCENTAGE)
DESCRIPTION
--| This operation sets the input percentage value in the Input/Output Descriptor related to the specified bargraph. The colour
    and flashing are the current one.|--
USED_OPERATIONS
    Input_Output_Driver.Put;
PROPAGATED_EXCEPTIONS
    NONE
HANDLED_EXCEPTIONS
    Input_Output_Driver.MALFUNCTION;
PSEUDO_CODE
    --| TBD |--
CODE
    --| TBD |--
END_OPERATION Display
OPERATION Set_Colour(BARGRAPH : in T_BARGRAPH;
                     COLOUR : in T_COLOUR)
DESCRIPTION
--| This operation displays the bargraph in the specified colour. The value and flashing are the current one.|--
USED_OPERATIONS
    Input_Output_Driver.Put;
PROPAGATED_EXCEPTIONS
    NONE
HANDLED_EXCEPTIONS
    Input_Output_Driver.MALFUNCTION;
PSEUDO_CODE
    --| TBD |--
CODE
    --| TBD |--
END_OPERATION Set_Colour
OPERATION Flash(BARGRAPH : in T_BARGRAPH;
                STATUS : in T_FLASHING)
DESCRIPTION
--| This operation sets the specified bargraph to flash. The colour and the value are the current one.|--
USED_OPERATIONS
    Input_Output_Driver.Put;
PROPAGATED_EXCEPTIONS
    NONE
HANDLED_EXCEPTIONS
    Input_Output_Driver.MALFUNCTION;
PSEUDO_CODE
    --| TBD |--
CODE
    --| TBD |--
END_OPERATION Flash
OPERATION Switch_Off
DESCRIPTION
--| This operation switches the hardware bargraphs off.|--
```

USED_OPERATIONS
Input_Output_Driver.Put;
PROPAGATED_EXCEPTIONS
NONE
HANDLED_EXCEPTIONS
Input_Output_Driver.MALFUNCTION;
PSEUDO_CODE
--| TBD |--
CODE
--| TBD |--
END_OPERATION Switch_Off
END_OBJECT *Bargraphs*

A1.3.4 OBJECT ALARM IS ACTIVE

DESCRIPTION
--| The Alarm object manages a set of software alarms. One hardware alarm is associated to those software alarms. It is possible to switch a software alarm on or off at each time. The hardware alarm is started when an unset software alarm is switched on (set). The hardware alarm is stopped when the set of software alarms are switched off (unset) or when acknowledged by the user. In that case the status of the set software alarms are not modified.|--
IMPLEMENTATION_CONSTRAINTS
--| 255 software alarms may be simultaneously.|--
PROVIDED_INTERFACE
TYPES
T_ALARM is integer range 1..255;
--| This type defines a software alarm, identified by an integer range1 to 255.|--
OPERATIONS
Start;
--| The Start operation switches the hardware alarm and the software alarms off. |--
Acknowledge;
--| This operation stops the hardware alarm without unset the set software alarms. |--
Switch_On (ALARM : in T_ALARM);
--| This operation switched the specified software alarm on. If this alarm was not yet set, the hardware alarm is started (if it was not). In the other cases, nothing is modified.|--
Switch_Off (ALARM : in T_ALARM);
--| The Switch_Off operation switched the specified software alarm off. If the hardware alarm is yet started and all the other software alarms are unset, the hardware alarm is stopped.|--
Stop;
--| The Stop operation switches the hardware alarm off.|--
OBJECT_CONTROL_STRUCTURE
DESCRIPTION
--| The Alarm accepts an Acknowledge operation only before a Stop and after a Start. The Acknowledge is not significant if the alarm is not previously set.|--
CONSTRAINED_OPERATIONS
Acknowledge CONSTRAINED_BY ASER_by_IT Ack_push-button;
REQUIRED_INTERFACE
OBJECT Input_Output_Driver;
TYPES
T_DEVICE;
T_DEVICE_STATUS;
T_DESCRIPTOR;
CONSTANTS
MAX_ALARMS_NUMBER;
OPERATIONS
Put;
EXCEPTIONS
MALFUNCTION;
DATAFLOWS

```
    data_out => Input_Output_Driver;
EXCEPTION_FLOWS
    malfunction <= Input_Output_Driver;
INTERNALS
OBJECTS
    NONE
TYPES
    T_SOFTWARE_ALARM_STATUS is (ON, OFF);
    --| Define the two possible status of a software alarm.|--
    T_HARDWARE_ALARM_STATUS is (ON, OFF);
    --| Define the two possible status of a hardware alarm.|--
    T_SOFTWARE_ALARMS is array (T_ALARM'RANGE) of
                            T_SOFTWARE_ALARM_STATUS;
    --| Define the set of software alarms.|--
    --| TBD |--
CONSTANTS
    --| TBD |--
OPERATIONS
    --| TBD |--
EXCEPTIONS
    --| TBD |--
DATA
    HARDWARE_ALARM : Input_Output_Driver.T_DEVICE (ALARM);
    HARDWARE_ALARM_STATUS : T_HARDWARE_ALARM_STATUS := OFF;
    SOFTWARE_ALARMS : T_SOFTWARE_ALARMS;
    --| TBD |--
OBJECT_CONTROL_STRUCTURE
PSEUDO_CODE
    --| TBD |--
CODE
    --| TBD |--
OPERATION_CONTROL_STRUCTURES
OPERATION Start
DESCRIPTION
--|  The Start operation switches the hardware and the software alarm off.|--
USED_OPERATIONS
    Input_Output_Driver.Put;
PROPAGATED_EXCEPTIONS
    NONE
HANDLED_EXCEPTIONS
    Input_Output_Driver.MALFUNCTION;
PSEUDO_CODE
    --| TBD |--
CODE
    --| TBD |--
END_OPERATION Start
OPERATION Acknowledge
DESCRIPTION
--|  This operation stops the hardware alarm.|--
USED_OPERATIONS
    Input_Output_Driver.Put;
PROPAGATED_EXCEPTIONS
    NONE
HANDLED_EXCEPTIONS
    Input_Output_Driver.MALFUNCTION;
PSEUDO_CODE
    --| TBD |--
CODE
    --| TBD |--
END_OPERATION Acknowledge
OPERATION Switch_On (ALARM : in T_ALARM)
DESCRIPTION
```

```
--|  This operation sets the specified software alarm to ON if it was not. If it was OFF, it starts the hardware alarm if it was not
    and sets the hardware alarm status to ON.|--
USED_OPERATIONS
    Input_Output_Driver.Put;
PROPAGATED_EXCEPTIONS
    NONE
HANDLED_EXCEPTIONS
    Input_Output_Driver.MALFUNCTION;
PSEUDO_CODE
    --| TBD |--
CODE
    --| TBD |--
END_OPERATION Switch_On
OPERATION Switch_Off (ALARM : in T_ALARM)
DESCRIPTION
--|  This operation sets the specified software alarm to OFF (if it was not). If all the software alarm status are OFF, it stops the
    hardware alarm if it was not and sets the hardware alarm status to OFF.|--
USED_OPERATIONS
    Input_Output_Driver.Put;
PROPAGATED_EXCEPTIONS
    NONE
HANDLED_EXCEPTIONS
    Input_Output_Driver.MALFUNCTION;
PSEUDO_CODE
    --| TBD |--
CODE
    --| TBD |--
END_OPERATION Switch_Off
OPERATION Stop
DESCRIPTION
--|  The Stop operation switches the hardware alarm off.|--
USED_OPERATIONS
    Input_Output_Driver.Put;
PROPAGATED_EXCEPTIONS
    NONE
HANDLED_EXCEPTIONS
    Input_Output_Driver.MALFUNCTION;
PSEUDO_CODE
    --| TBD |--
CODE
    --| TBD |--
END_OPERATION Stop

END_OBJECT Alarm
```

A2 EXAMPLE OF ADA CODE IMPLEMENTATION

```
-- OBJECT EMS IS ACTIVE
-- INTERNAL OBJECTS : Ctrl_EMS, Alarm, Sensors, Bargraphs
with Ctrl_EMS, Alarm;
package EMS is
-- DESCRIPTION
The EMS monitors and displays the oil pressure, water temperature and fuel level on the appropriate bargraphs, and starts an
    alarm in case of a value is out of a predefined range. First, the EMS shall be started. It may be stopped at any moment.
-- IMPLEMENTATION_OR_SYNCHRONISATION_CONSTRAINTS
Start, Stop and Acknowledge operations are mutually exclusive (each of those operations shall wait the end of an other one
    before to be started).
-- PROVIDED_INTERFACE
   -- OPERATIONS
   procedure Start renames Ctrl_EMS.Start;
The Start operation starts the EMS. The EMS environment will be initialised.
   procedure Stop renames Ctrl_EMS.Stop;
The Stop operation stops the EMS.
   procedure Acknowledge renames Alarm.Acknowledge;
The Acknowledge operation stops the Alarm for the faults which have switched on it.
-- OBJECT_CONTROL_STRUCTURE
   -- DESCRIPTION
The EMS accepts start, stop and acknowledge operations at any time. Start is not significant after a start. Stop or acknowledge
    are not significant after a stop.
   -- CONSTRAINED_OPERATIONS
      -- Start CONSTRAINED_BY ASER_by_IT Start_push-button
      -- Stop CONSTRAINED_BY ASER_by_IT Stop_push-button
      -- Acknowledge CONSTRAINED_BY ASER_by_IT Ack_push-button
   -- IMPLEMENTED_BY Ctrl_EMS, Alarm
end EMS;


-- OBJECT Ctrl_EMS IS ACTIVE
package Ctrl_EMS is
-- DESCRIPTION
--  This object is the controller of the EMS. It starts and stops all the constituents objects of the EMS and controls the moni-
    toring.
-- IMPLEMENTATION_OR_SYNCHRONISATION_CONSTRAINTS
--  The Ctrl_EMS must perform the monitor operation in less than 1 second.

-- OBJECT_CONTROL_STRUCTURE
   -- DESCRIPTION
--  The Ctrl_EMS accepts start, stop and monitor operations at any time. A start is not significant after a start. A stop or a mon-
    itor --are not significant after a stop.
   -- CONSTRAINED_OPERATIONS
      -- Start CONSTRAINED_BY ASER_by_IT Start push-button
      -- Stop CONSTRAINED_BY ASER_by_IT Stop push-button
      -- Monitor CONSTRAINED_BY ASER_by_IT Timer_1Hz

      task OBCS_Ctrl_EMS is
         entry Start;
         entry Stop;
         entry Monitor;
      end OBCS_Ctrl_EMS;

-- PROVIDED_INTERFACE
   -- OPERATIONS
   procedure Start renames OBCS_Ctrl_EMS.Start;
```

```
--The Start operation initialises the monitoring timer, the alarm, the sensors and the bargraphs which are used during the monitoring.
   procedure Stop renames OBCS_Ctrl_EMS.Stop;
--The Stop operation stops the monitoring timer and switches off the alarm and the bargraphs and stops the sensors.
   procedure Monitor renames OBCS_Ctrl_EMS.Monitor;
--The Monitor operation acquires the values of each sensor. Those values are displayed on the appropriate bargraph. In case of a value --is out of range, this operation switches the alarm on and gives the red colour to the bargraph. If there is a sensor failure, the alarm --is switched on and the bargraph flashes in red colour. In other cases, the values are displayed in green colour.

-- DATAFLOWS
   -- mean_value <= Sensors
   -- mean_value => Bargraphs
   -- color => Bargraphs
   -- initial_frequency => Timers_Driver

-- EXCEPTION_FLOWS
   -- sensor_failure <= Sensors

-- END_OBJECT Ctrl_EMS
end Ctrl_EMS;
-- OBJECT Ctrl_EMS IS ACTIVE
-- REQUIRED_INTERFACE
   -- OBJECT Sensors
   with  Sensors;
      -- TYPES
            -- T_SENSOR
      -- OPERATIONS
            -- Start
            -- Acquire
            -- Stop
      -- EXCEPTIONS
            -- SENSOR_FAILURE
   -- OBJECT Alarm
   with  Alarm;
      -- TYPES
            -- T_ALARM
      -- OPERATIONS
            -- Start
            -- Switch_On
            -- Switch_Off
            -- Stop
   -- OBJECT Bargraphs
   with  Bargraphs;
      -- TYPES
            -- T_COLOUR
            -- T_BARGRAPH
            -- T_FLASHING
            -- T_PERCENTAGE
      -- OPERATIONS
            -- Init
            -- Display
            -- Set_Colour
            -- Flash
            -- Switch_Off
   -- OBJECT Timers_Driver
   with Timers_Driver;
      -- REQUIRED TYPES
            -- T_TIMER
      -- REQUIRED OPERATIONS
            -- Create
            -- Start
```

```
            -- Delete
package body Ctrl_EMS is
-- INTERNALS
   -- TYPES
      -- TBD
   -- CONSTANTS
      IT_1HZ_ADDRESS : constant := Monitor'ADDRESS;
      MONITORING_FREQUENCY : constant := 1;
      -- TBD
   -- OPERATIONS
      procedure Is_Value_out_of_Range;
      procedure OPCS_Start;
      procedure OPCS_Stop;
      procedure OPCS_Monitor;
      -- TBD
   -- EXCEPTIONS
      -- TBD
   -- DATA
      MONITORING_TIMER : Timers_Driver.T_TIMER;
      -- TBD

-- OBJECT_CONTROL_STRUCTURE
   -- PSEUDO_CODE
      -- TBD
   -- CODE
      task body OBCS_Ctrl_EMS is
      begin
         loop
            loop
               select
                  accept Start; OPCS_Start; exit;
               or
                  accept Stop; -- empties Stop queue
               or
                  accept Monitor; -- empties Monitor queue
               end select;
            end loop;
            loop
               select
                  accept Start; -- empties Start queue
               or
                  accept Stop; OPCS_Stop; exit;
                  -- only when monitoring is completely finished
               or
                  accept Monitor; OPCS_Monitor;
               end select;
            end loop;
         end loop;
      end OBCS_Ctrl_EMS;

-- OPERATION_CONTROL_STRUCTURES

   -- OPERATION OPCS_Start
   procedure OPCS_Start is
   -- DESCRIPTION
--This operation initialises the system (the alarm, the bargraphs, the sensors) and then creates and starts a timer for it triggers
    the mon--itoring every 1 second.
   -- USED_OPERATIONS
      -- Timers_Driver.Create
      -- Timers_Driver.Start
      -- Sensors.Start
      -- Alarm.Start
```

```
      -- Bargraphs.Init
   -- PSEUDO_CODE
      -- TBD
   -- CODE
      begin
         Timers_Driver.Create (MONITORING_TIMER,
                               MONITORING_FREQUENCY,
                               IT_1HZ_ADDRESS );
         Alarm.Start;
         Bargraphs.Init;
         Sensors.Start;
         Timers_Driver.Start ( MONITORING_TIMER );
Monitoring Timer is started only when all the hardware devices are initialised.
      end OPCS_Start;
   -- END_OPERATION OPCS_Start

   -- OPERATION OPCS_Stop
   procedure OPCS_Stop is
   -- DESCRIPTION
--This operation stops all the system (the alarm, the bargraphs, the sensors) and resets the monitoring timer.
   -- USED_OPERATIONS
      -- Timers_Driver.Delete
      -- Alarm.Stop
      -- Bargraphs.Switch_Off
      -- Sensors.Stop
   -- PSEUDO_CODE
      -- TBD
   -- CODE
      begin
         Timers_Driver.Delete ( MONITORING_TIMER );
         Sensors.Stop;
         Alarm.Stop;
         Bargraphs.Switch_Off;
      end OPCS_Stop;
   -- END_OPERATION OPCS_Stop

   -- OPERATION Is_Value_out_of_Range
   procedure Is_Value_out_of_Range is
   -- DESCRIPTION
      -- TBD
   -- USED_OPERATIONS
      -- TBD
   -- PROPAGATED_EXCEPTIONS
      -- TBD
   -- HANDLED_EXCEPTIONS
      -- TBD
   -- PSEUDO_CODE
      -- TBD
   -- CODE
      begin
         null;
         -- TBD
      end Is_Value_out_of_Range;
   -- END_OPERATION Is_Value_out_of_Range

   --OPERATION Monitor
   procedure OPCS_Monitor is
   -- DESCRIPTION
--This operation is in charge of the EMS monitoring. It acquires the mean values of each sensor, controls them against limit
   values --with respect to each category of sensors, converts those mean values into displaying values according to min. and
   max.  infor--mation and displays them on the corresponding bargraph. It sets the alarm if the mean value is out of range or
   if there is a sensor --failure. It also set the green colour of a bargraph when the value is correct, the red colour when the
```

```
      value is out of range and the --red flashing colour when  the corresponding sensor is failed.
      --The Monitor operation switched the alarm on for each out of range value and each failed sensor and switched the alarm
      off for --each correct value and running sensor.
      -- USED_OPERATIONS
         -- Alarm.Switch_On
         -- Alarm.Switch_Off
         -- Bargraphs.Display
         -- Bargraphs.Flash
         -- Bargraphs.Set_Colour
         -- Sensors.Acquire
         -- Is_Value_out_of_Range
      -- HANDLED_EXCEPTIONS
         -- Sensors.SENSOR_FAILURE
      -- PSEUDO_CODE
         -- TBD
      -- CODE
         begin
               null;
               -- TBD
         end OPCS_Monitor;
      -- END_OPERATION OPCS_Monitor
   -- END_OBJECT Ctrl_EMS
   end Ctrl_EMS;
```

A3 HOOD4 TARGET IMPLEMENTATION AND ILLUSTRATIONS

A3.1 HOOD4 TO TARGET IMPLEMENTATION PRINCIPLES

By convention, there is one OBCS per active object,which shall handle all dynamivc process/ task ineteractions associated to the dynamic (i.e. the way the operations are executed) behaviour of the object. In case of resource shortage, its is always possible to group several OBCS into one.

HOOD standard generation rules [HRM] define the OBCS as an Ada package(nested or not in the package associated to the current object), possibly reduced to one OBCS task unit, and the OPCS header code as rename of an OBCS entry hiding the OBCS to clients, whereas the OPCS body code is defined in a procedure called OPCS_<OP_NAME

This solution, the standard code generation schema in HOOD3 is however not always matching the needs of real time programming; for example one may want to execute an operation in mutual exclusion. A solution for such problems has been proposed to the HTG and may be included in the next evolution of HOOD, named HOOD4. This solution is briefly outlined below.

In order to solve the problems associated to non supported execution request and protocols, and based on several attempts to conceptualize and formalize the behavior of process upon constrained operations, **a general solution has been achieved which splits the code associated to an OPCS into several LOGICAL PARTS which redefine the OBCS code.** Code generation rules are then easily derived by defining the source code (which may be empty) associated to each LOGICAL PART and depending on the type of constraints attached to an operation:

- OPCS_ERcode associated to an execution request for protocol constrained operation
- OPCS_HEADER code associated to a protected and/or state constrained operation
- OPCS_BODY the real functional code of the operation
- OPCS_FOOTER code associated to a protected constrained
- OBCS: client part executed code for queuing and execution request and waiting return parameters
- OBCS: server part executed code for dequeing Op requests, processing them and return parameters

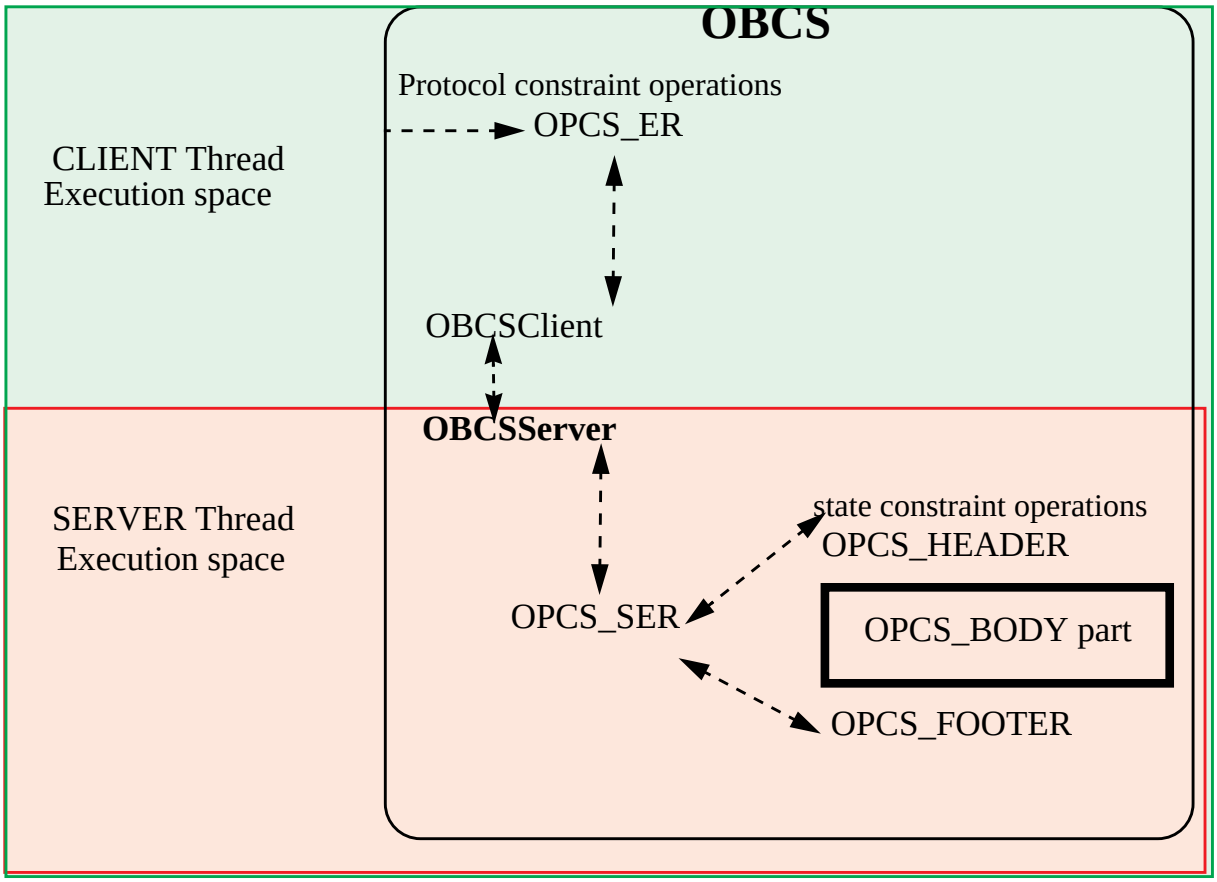


Figure 160 - Suggested Target Code structure for constraint operations

A3.1.1 IMPLEMENTING STATE CONSTRAINTS WITH STANDARD ADA

Constraints on operations may most of the time be described using a state transition diagram *where states are those of the object and transitions only triggered by operation execution.* The OBCS is then reduced to a correct implementation of the state diagram which is defined as an Ada task according to standard generation rules[HRM]. section 2.13.2 gives a full illustration of the Ada code for a the STACKS object .

A3.1.2 IMPLEMENTING STATE CONSTRAINTS WITHOUT ADA TASKING

In case of resources shortage or when Ada tasking should not be used, an alternative *implementation, based on Finite State Machines (FSM),* may be used, simulating the code structure as illustrated in figure above (up to standardization within HOOD and support by the toolsets). The OPCS code shall then be structured in three parts:

- OPCS_HEADER part for handling state constraints. The associated header code as illustrated in Figure 147 - is reduced to a simple call to an FSM module. This code attached to the

constraint resolution generally needs not an additional process: a simple test may implement the state constraint. Thus we implement the constraint using an FSM. The code of that FSM could be generated automatically from an appropriate STD, so that the designer has just to express the possible sequences of an object/class by drawing a STD. See *Appendix A3.2.1.2* for an example of such code>
Additionaly the OPCS_HEADER could contain code to set a semaphore when the operation is to be executed in mutual exclusion. (this is done by a call to the target OS services.)

- Core body part: containing the functional, transformational code, not dealing with states and behaviours
- OPCS_FOOTER part: code needed only to release a semaphore in case of mutual exclusion constraint.

The semantics of *state constrained operations* of HOOD[HRM] is thus enforced, since the return to the client is made either with the OPCS executed (the state was OK at the operation request) or with OPCS non executed (the state was not OK at calling time). *Appendix A3.2* gives a full illustration of such ADA code.

A3.1.3 IMPLEMENTING PROTOCOL CONSTRAINTS IN ADA

Protocol constraint operations are specified through trigger labels expressed in text structured with keywords such as HSER, LSER, TOER_HSER, TOER_LSER or ASER, that may apply together with state constraints. These protocols, specified in [HRM] are used to express inter-process communications, as well as inter-node communication in case of virtual nodes defining distributed code or memory partitions.
Using the queuing and mutual exclusion mechanisms supported by the Ada **select** and **accept** statements, standard code generation rules[HRM] for constraint protocols are rather straightforward**,** but always involving a server OBCS task, and often with the need of copying parameters inside the **accept** statement. An example of such generated code is given in section 2.13.2

A3.1.3.1 Implementing protocol constraints without Ada tasking

In case of resources shortage or when Ada tasking should not be used, generation rules again rely on queuing, synchronisation and message passing mechanisms, but this time using classical operating services. We hope that in the near future this code will also be standardised, using efficient communication standards such as those promised between *threads* and already available on the most recent UNIX releases. Figure 161 - gives the HOOD execution model associated to a protocol constrained implementation, where:

- the code executed by the client thread comprises:
  - the OPCS_ER (see below) and
  - the **ClientObcsPart** which sends a request MSG to the server process and waits for a return MSG
- the code executed by the server thgread comprises:
  - a **ServerObcsPart** which waits for a request MSG, and dispatches the request to the associated local operation, that is
  - the OPCS_SER which executes the effective OPCS body code, and returns back a return

MSG to the client .

When the queuing mechanisms used also supporting distribution[1], this solution can be directly used for VN code generation. In that case ClientObcs parts may be grouped into a common VNCS module. (VN Control Structure)

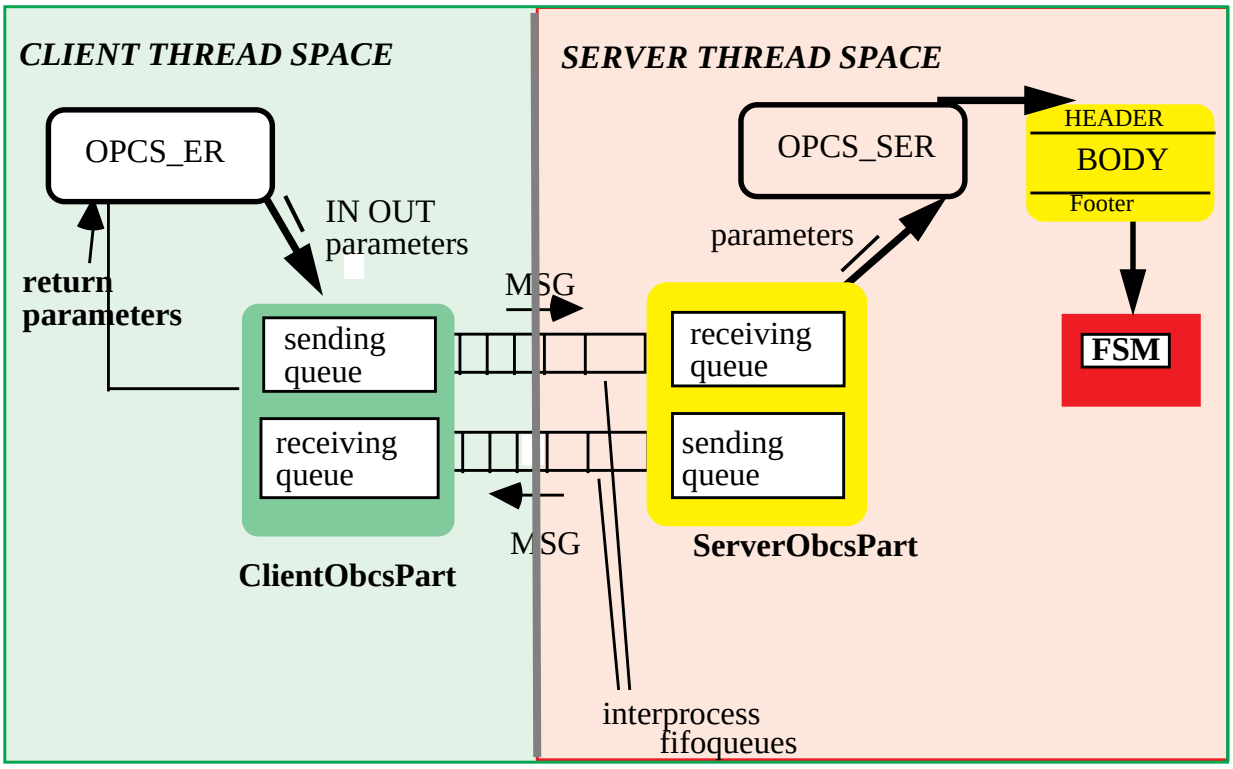


Figure 161 - Principle of Code for Protocol onstrained operation support

Since such a schema is not yet supported by HOOD tools, the code for OPCS_ER, OPCS_SER and client part and server parts should be structured in such a way as to isolate the original functional code. This could be done by defining for each active object *<NAME>*:

- the original object of name *<NAME>_SERVER* where constrained operations are reduced to state constrained operation s and OPCS structured into HEADER, BODY and FOOTER parts.
- a client code part as an object named *<NAME>* providing unconstrained operation, with same signatures as the protocol constrained one from the original object, and having OPCS implementing OPCS_ER code
- a "request_broker" code part as an object named <NAME>_RB providing unconstrained operations, with same signatures as the protocol constrained one from the original object, and having OPCS implementing OPCS_SER code, which require operations provided by *<NAME>_SERVER*

Such strategy is illustrated in Figure 162 - below.

1. generic pacakge BUFFERS of the EXTRA proposal [ISO/JTC1/SC22/WG9/RRG} [Work Item Number JTC 1.22.35, 31/03/93] gives already specifications of such code..

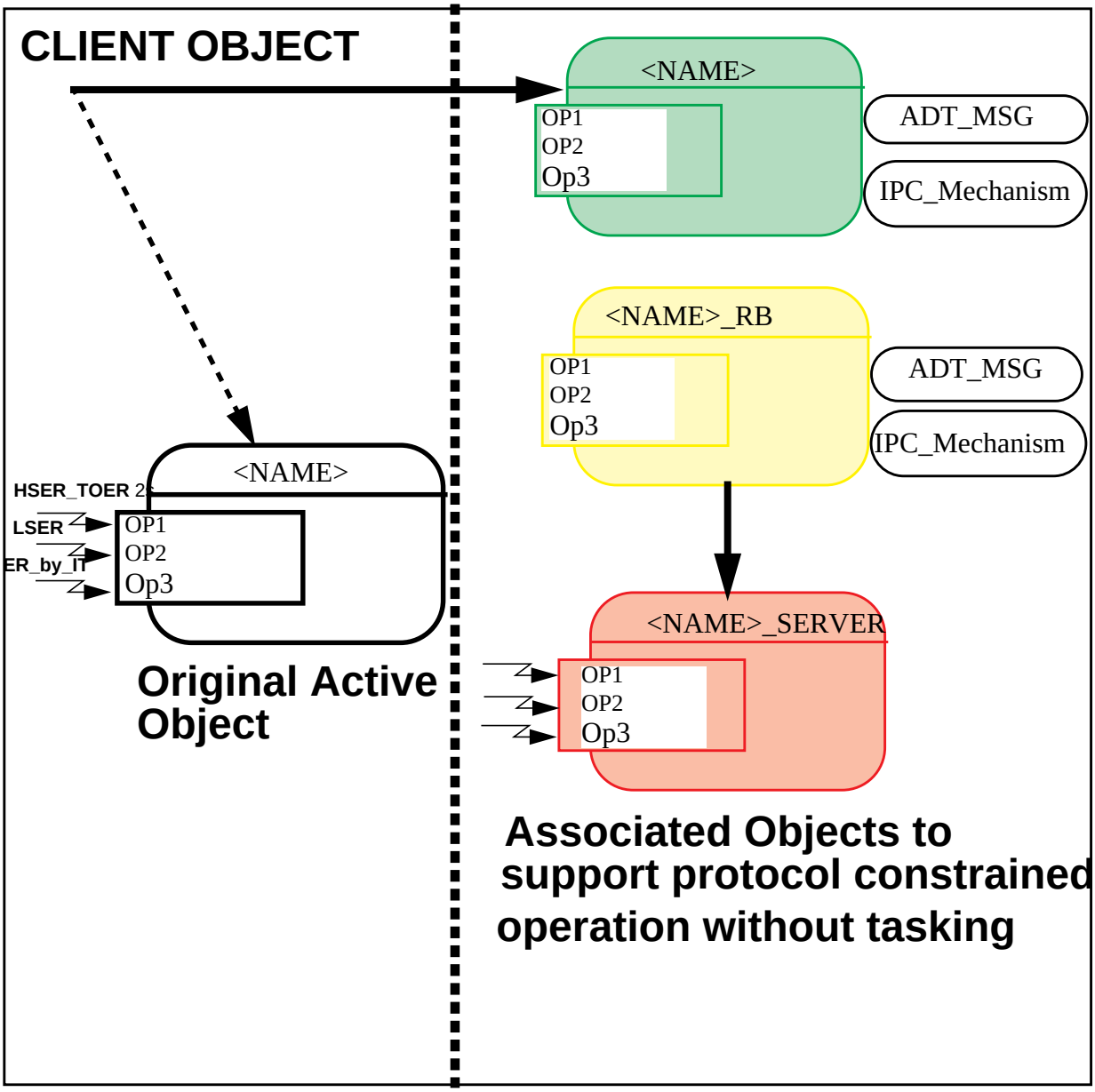


Figure 162 - Principle of Code for Protocol Constrained operation support

A3.2 HOOD4 ADA CODE ILLUSTRATION

A3.2.1 HOOD4 STATE CONSTRAINT SUPPORT

The approach taken for enforcing the state of an object or a class is to raise an exception[2] if the state was not OK for the operation to execute. Figure 163 - illustrates the associated target procedure code associated to a state constrained operation.

```
procedure <State_Constrained_Operation>  is -- target code structure
begin
-- OPCS_HEADER part (automatically generated)------------------------------
    HRTS.SEMAPHORE.P(<object Name>); -- seize MUTEX Semaphore
    HRTS.FSMs.FIRE(operation);      -- try to fire  operation transition
    -- if execution allowed in current state, exception BAD_EXECUTION REQUEST is raised.
    -- end OPCS_HEADER ------------------------------
-------------------------------------------------
    OPCS BODY CODE ; -- extracted directly from ODS fields
-------------------------------------------------
-- OPCS_FOOTER part (automatically generated)------------------------------
    HRTS.SEMAPHORE.V(<object Name); -- release MUTEX Semaphore
    -- end OPCS_FOOTER ------------------------------
end;
```

Figure 163 - State Constraints Implementation Schema

A3.2.1.1 STACK OSTD

This seems well founded and pragmatic, since automatically enforcing a state according to a specification given in an Object State Transition Diagram (OSTD) as illustrated in Figure 164 - could lead to uncontrollable and less understandable[3] code.

.

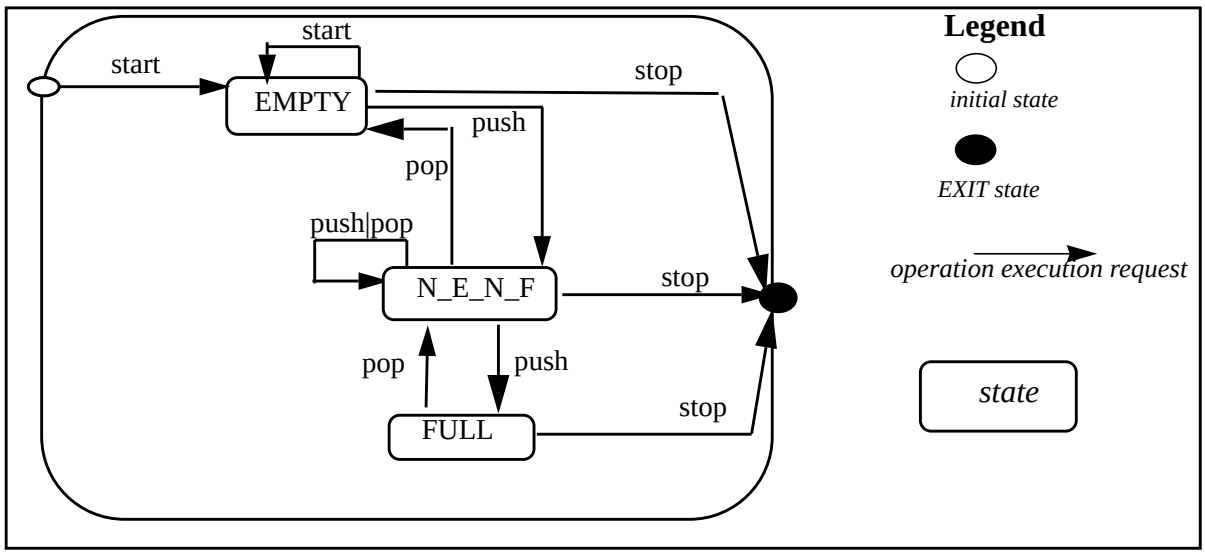


Figure 164 - OSTD for STACK object

[2] (to abort the operation request)

[3] (see the paper [18] showing the difficulty of enforcing correct state (wrt a state specification)).

A3.2.1.2 STACK OSTM

Figure 165 - illustrates the OSTM (Object State Machine), an implementation of the STACK OSTD which could be generated automatically from the OSTD by a HOOD toolset.
Such code is standardised using the HRTS.FSM module (Finite State Machine where transitions are only triggered by operation execution requests) associated to an object or class instance. This FSM is initialised either at initialisation or when procedure FIRE for that object FSM is called for the first time[4].

```
with HRTS; use HRTS;  -- visibility to FSMs handling module of HRTS
procedure OSTM is -- OSTM for stack
    HRTS.FSMs.CREATE (STACK,4,3,EMPTY); //4 possible ER and 3 states, initial state is EMPTY
    HRTS.FSMs.TRANS(STACK, EMPTY,Start,EMPTY);
    HRTS.FSMs.TRANS(STACK, EMPTY,Push,NON_E_F);
    HRTS.FSMs.TRANS(STACK, EMPTY,Stop,EXIT);
    HRTS.FSMs.TRANS(STACK, NON_E_F,Start,EMPTY);
    HRTS.FSMs.TRANS(STACK, NON_E_F,Push,NON_E_F);
    HRTS.FSMs.TRANS(STACK, NON_E_F,Pop,NON_E_F);
    HRTS.FSMs.TRANS(STACK, NON_E_F,Pop,EMPTY); -- if stack .top =1
    HRTS.FSMs.TRANS(STACK, NON_E_F,Stop,EXIT);
    HRTS.FSMs.TRANS(STACK, NON_E_F,Push,FULL); -- if stack.top=max-1
    HRTS.FSMs.TRANS(STACK, FULL,Pop,NON_E_F);
    HRTS.FSMs.TRANS(STACK, FULL,Start,EMPTY);
    HRTS.FSMs.TRANS(STACK, FULL,Stop,EXIT);
    end;
```

Figure 165 - OSTD Implementation Example

A3.2.1.3 STACK with state constraints in Ada

For conciseness we take the STACK object example for illustrating the associated target code[5] in Ada95[6]. STACK has operations PUSH and POP HSER constrained, and operation STOP state and ASER constrained, and operation START only state constrained, semantically correct only when the behaviour expressed in OSTD in Figure 163 - is enforced :

[4] Any suggestion that works reliably both in ADA and c++ and for both objects and classes very well come!

[5] Target ADA units are represented as squared boxes, with its kind and name represented in the top, and required units represen-tented as attached small boxes.

[6] Translation into C++ is straight forward since we do not use here any "tasking feature". As "thread supporting UNIX targets are available, translation from ADA95to C+ would also become straight..

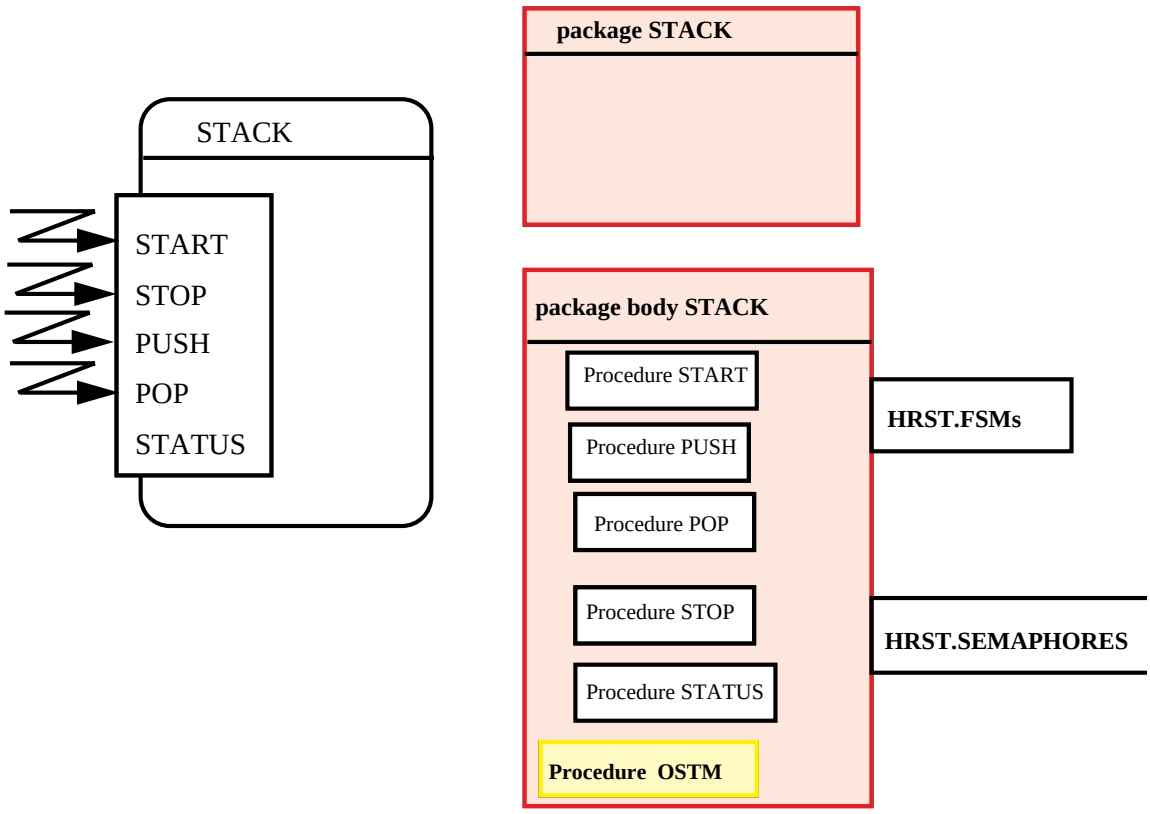


Figure 166 - STACK object with state constraints operations

```
with Items; use Items; -- visibility on STACK items
package STACK is
    procedure start;  -- only state constraint
    procedure stop ;
    procedure PUSH(item : in T_Item) ;
    procedure POP(Item :out T_Item);
    function Status return T_Status ;
end STACK;
```

Figure 167 - STACK with state constraints code Sample

```
package body STACK is -- any common data to be included here ------------------
    with HRTS; use HRTS;
    procedure OSTM is -- code as illustrated in  figure A1.1 above
    end OSTM;
    procedure start is separate;
    procedure stop is separate;
    procedure PUSH(item : in T_Item)is separate;
    procedure POP(Item :out T_Item)is separate;
    function Status return T_Status is separate;
begin
    OTSM; -- initialisation of FSM at package elaboration[7]
end STACK;-------------------------------------------------
separate (STACK)
procedure start is              -- begin OPCS_HEADER ------------------------------------------------------
    HRTS.SEMAPHORE.P(STACK); -- there is one  mutex semaphore for STACK
    begin
    HRTS.FSMS.FIRE(STACK,START);--X_BAD_EXECUTION_REQUEST possibly  raised
    exception
    when X_BAD_EXECUTION_REQUEST =>
       EXCEPTIONS.LOG("STACK.Start", "X_BAD_EXECUTION_REQUEST");
       EXCEPTIONS.raise;  --shall work in ADA and in  C++
    when Others =>
       EXCEPTIONS.LOG("STACK.Start", "Others");
       EXCEPTIONS.raise;-- so as to propagate to client
    end;                                -- end OPCS_HEADER ------------------------------------------------------
--  OPCS_BODY_CODE; -- extracted from the ODS field;
-- begin OPCS_FOOTER ------------------------------------------------------
    HRTS.SEMAPHORE.V(STACK); -- release  mutex semaphore for STACK
-- end OPCS_FOOTER --------------------------------------------------------
end start;-------------------------------------------------

separate (STACK)
procedure PUSH(item : in T_Item) is -- begin OPCS_HEADER ----------------------------------------------------
    HRTS.SEMAPHORE.P(STACK); --  mutex still need since another client may start, stop in the meantime.
    begin
    HRTS.FSMS.FIRE(STACK,PUSH);--  X_BAD_EXECUTION ERQUEST possibly raised
    exception
    when X_BAD_EXECUTION_REQUEST =>
       EXCEPTIONS.LOG("STACK.Push", "X_BAD_EXECUTION_REQUEST");
       EXCEPTIONS.raise;  -- so as to propagated to client
    when Others =>
       EXCEPTIONS.LOG("STACK.Push", "Others");
       EXCEPTIONS.raise;
    end;                                -- end OPCS_HEADER ------------------------------------------------------
-- OPCS_BODY_CODE; --extracted from the ODS field;
     if TOP=max then-- possibly force object 's  state to FULL
       HRTS.FSMS.SET(STACK, FULL);-- where OPCS code can still interact on state
     end if;
-- begin OPCS_FOOTER ------------------------------------------------------
    HRTS.SEMAPHORE.V(STACK); -- release mutex semaphore for STACK
-- end OPCS_FOOTER --------------------------------------------------------
end PUSH;
function Status is  -- non constrained operations have empty  HEADER or FOOTER code.
begin
    [STACK STatus]
end STatus;
```

Figure 168 - STACK with state constraints code Sample

Comments:

- Including OPCS_BODY directly in the procedure bodies avoids efficiency losses with pa-

[7]. When used for active class instances, the contructor (in ADA and C++) shall have to call the OSTM.

rameter pushing on and back memory frames.

- MUTEX semaphores are needed for ensuring state integrity in a concurent context, and **are generated by default.** Optional rules shall specify under which conditions they can be suppressed (for e.g. if all operations are HSER or LSER[8], in which case the SERVEROBCS warrants mutual exclusion because it is its own flow executing OPCS_OP code.)

A3.2.2 PROTOCOL CONSTRAINTS SUPPORT ILLUSTRATION

Protocol constraints implementation leads to interprocess communication (pure synchronisation and/or data communication) between at least one client process and one server process which execute each in their own memory partition. The target architecture principle applied to the STACK example with protocol constrained operations leads to the following Ada code structure

For one active object, the target code is structured into three packages of which code samples are given below. This code structure may be directly mapped in C++ modules and shall be optimised when using Ada tasking or thread supported OS; *marshalling* [10] in OPCS_ER code and *unmarshalling* in OPCS_SER code as well as InterProcessCommunication (IPC) message handling is not needed when threads or task share parameters in the same memory partition

8. Should probably be supported by ADA code generators, since users are hungry for efficient code.

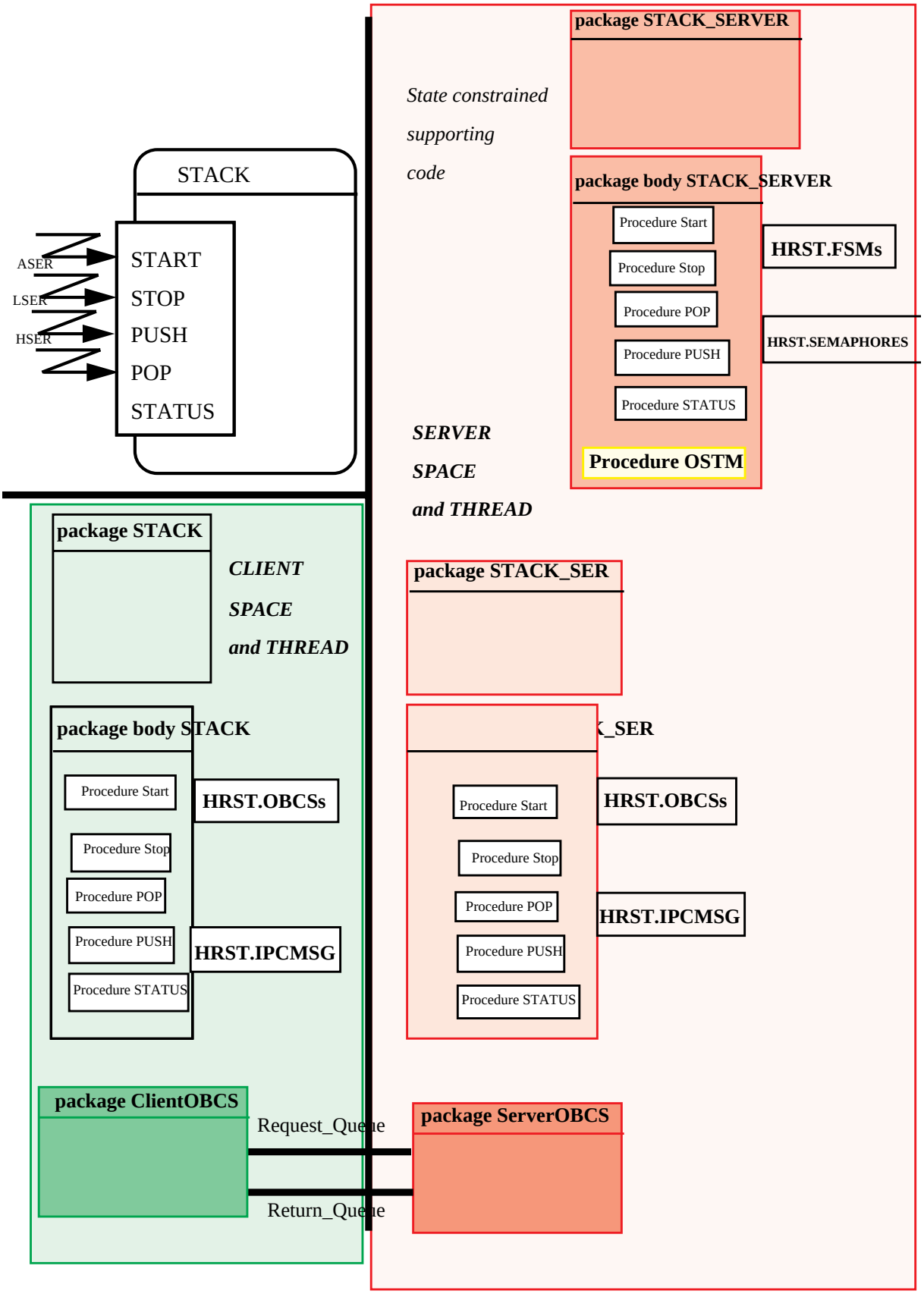


Figure 169 - STACK with protocol constraints

A3.2.2.1 STACK with protocol constrained operations (Client code)

```
with Items; use Items;  -- package spec. as for passive object
package STACK is
    procedure start;
    procedure stop;
    procedure PUSH(item : in T_Item);
    procedure POP(Item :out T_Item);
    function Status return T_Status;
end STACK;
```

Figure 170 - STACK with protocol constraints (without tasking)

```
with HRTS.IPCMSG; -- InterProcessCommunication  Message Handling
with HRTS.OBCSs; -- Module for Client And Server OBCS code
package body STACK is
-- not any data here since all deported in the STACK_SERVER object in server partition----------
    package ClientObcs is new HRTS.OBCSs.ClientObcs (STACK,... );
    procedure PUSH(item : in T_Item) is -- OPCS_ER code
    MSG : IPCMSG.T_MSG;
    begin
        MSG=IPCMSG.create;  - initialise an IPC MSG data structure
        MSG.sender=STACK;
        MSG.OPERATION=PUSH;
        MSG.CNSTRNT=HSER[9];
        MSG.INFO=Item;
        ClientObcs.insert(MSG); -- insert in request queue
        MSG:=ClientObcs.remove; -- remove return IPCMSG
        if not (MSG.X=OK)then
            EXCEPTIONS.raise(MSG.X); --raise exception according to Xvalue
        else
            EXCEPTIONS.LOG ("STACK.PUSH_ER", "OK return from server")
        end if;
        if not (MSG.CSTRNT=ASER) then
            ClientObcs.FREE(MSG); -- deallocate MSG and queues
        end if;
    exception
    when X_BAD_EXECUTION_REQUEST =>
        EXCEPTIONS.LOG("STACK.PUSH", "X_BAD_EXECUTION_REQUEST");
        EXCEPTIONS.raise;  -- so as to propagated to client
    when Others =>
        EXCEPTIONS.LOG("STACK.PUSH", "Others");
        EXCEPTIONS.raise;
--similar code for POP,STOP and START and Status
    procedure start is separate;
-- code of state constrained operation are treated as HSER on client side.HSER is needed in order to get back the results of
    FSM firing.
    function Status return T_Status is separate;
-- non constrained operation become HSER constrained operations since return parameters must be accessed from the
    <OBJECT>_SERVER code.
```

Figure 171 - STACKwith protocol constraints (without tasking)

[9] if the request is also TOER, then we would insert here a call to TIMER.SET(delay), as well as a TIMER.TimeOut call after return (MSG:=ClientObcs.remove; -- remove return IPCMSG

A3.2.2.2 STACK with protocol constrained operations (STACK_RB CODE)

```
with Items; use Items;
with STACK_SERVER; -- visibility need to have access to real operations and data.
package STACK_RB is
    procedure start renames STACK_SERVER.Start;
    procedure stop;
    procedure PUSH(item : in T_Item);
    procedure POP(Item :out T_Item);
    function Status return T_Status renames STACK_SERVER.Status;end STACK_SER;
end STACK_SER;

with HRTS.IPCMSG; -- InterProcessCommunicationMEssage Handling
with HRTS.OBCSs; -- Module for Client And Server OBCS code
package body STACK_SER is

    procedure PUSH (MSG : in IPCMSG.T_MSG) is -- OPCS_SER code
    Item : Items.T_Items;
    begin
       begin
            item:= MSG.INFO;
            MSG.X=OK;
            if MSG.CSTRNT=LSER|LSER_TOER then
                ServerObcs.insert(MSG);  -- insert in return queue
            end if;
            STACK_SERVER.PUSH (Item);
       exception
       when X_BAD_EXECUTION_REQUEST =>
            EXCEPTIONS.LOG("STACK_SER.PUSH", "X_BAD_EXECUTION_REQUEST");
            MSG.X= string (X_BAD_EXECUTION_REQUEST);
       when Others =>
            EXCEPTIONS.LOG("STACK_SER.PUSH", "Others");
            MSG.X= string (OTHERS);
       end; -- of begin
       if MSG.CSTRNT=HSER|HSER_TOER then
            ServerObcs.insert(MSG); -- insert in return queue
       end if;
    end PUSH;
--similar code for POP_SER and STOP_SER

    procedure start is -- OPCS_SER code, also for STatus
    begin
       begin
            MSG.X=OK; STACK_SERVER.start;
       exception
       when X_BAD_EXECUTION_REQUEST =>
            • • • •-- exceptions handling as in procedure PUSH
       end; -- of begin
       ServerObcs.insert(MSG); -- insert in return queue as treated as HSER
    end start;


    procedure SerDispatcher(Name : T_Operation)  is \---
       Local _STACK: T_STACK; -- local Data for client-server code
       Local_Element : ADT_ELMENT.T_ELEMENT;
    begin
    case Name is  --
       when PUSH=>
            begin
                PUSH;
            exception
            when •••=>MSG.X.VALUE=X_•••
```

```
            when X_UNW=> MSG.X_VALUE=X_UNW;
            when others=>EXCEPTIONS.LOG("SerDispatcher, OPCS_PUSH", "Others");
                    MSG.X_VALUE=X_OTHERS; end ;
        when POP=>
            begin
            POP;
            exception
            when X_UNW=> MSG.X_VALUE=X_UNW;
            when others=>EXCEPTIONS.LOG("SerDispatcher, OPCS_POP", "Others");
                    MSG.X_VALUE=X_OTHERS; end;
        when others => EXCEPTIONS.LOG("  OBCS.OPCS.NAME", "NO SUCH NAME");
    end case;
    exception
        when others => EXCEPTIONS.LOG(" OBCS.OPCS.NAME", "Others");raise:
    end SerDispatcher;

procedure STACKSERVER is  -- object'IPC message server
    MSG : IPCMSG.T_MSG;-- local MSG data
    begin
    OBCS.INIT_POOL ;-- init of QUEUE pool
    loop
        QUEUES.Remove(R_Q,MSG);  --get a request from R_Q Queue
         case MSG.CSTRNT is  --process the MSG
            when  HSER| HSER_TOER =>
                SerDispatcher(MSG.OPERATION); -- execution du code concerné
                QUEUES.Insert(MSG.RTN_Q,MSG); -- renvoi MSG au client
            when LSER |LSER_TOER=>
                QUEUES.Insert(MSG.RTN_Q,MSG); -- renvoi MSG au client
                SerDispatcher(MSG.OPERATION); -- execution du code concerné
            when ASER => SerDispatcher(MSG.OPERATION); -- execution du code concerné
            when others =>EXCEPTIONS.LOG(" STACK OBCS INIT PROCESS LOOP", "BAD constraints")
                raise X_MSG_INCONSISTENCY;
        end case;
    end loop;
    exception  when others => EXCEPTIONS.LOG(" STACK OBCS INIT PROCESS LOOP", "EXCEPTION.NAME");
end START;
    end STACKSERVER;
```

Figure 172 - STACK 1 with protocol constraints (without tasking)

Note how the protocol constraints are solved :

- ASER constraint are treated directly in the STACKSERVER loop before the operation request is processed.
- TOER requests are treated at the client side using only local time.
- LSER request are treated as incoming parameters have been extracted.
- HSER request are treated after <OBJECT>_SERVER.operatiojn execution.

A3.2.2.3 STACK with protocol constrained operations (STACK_SERVER CODE)

same code as for state constrained STACK

A3.2.3 CLIENT_SERVER ILLUSTRATION FOR CLASSES

The above principles are adapted for defining code generation rules for an active class of name <CLASS>, to whom three target class units are associated : client_class module of name *<CLASS>*, request broker for the class of name *<CLASS_RB>* and the original class named *<CLASS>_server*.
Client of such classes shall only instanciate *client_class* target unit, as illustrated inFigure 173 - , with one server instance of *server_class* for all instances of a given client, thus enforcing Client_Server architecture principles

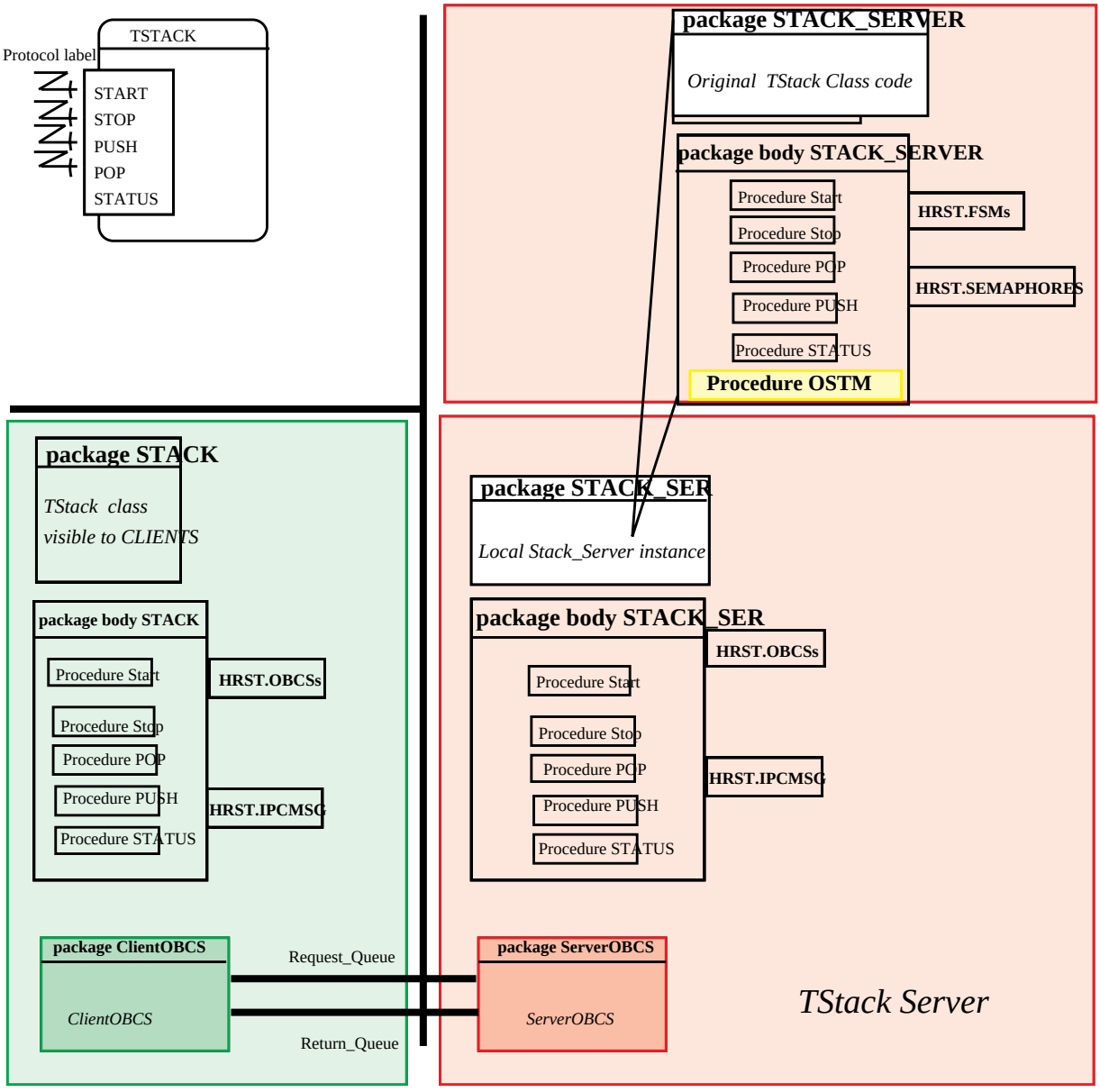


Figure 173 - Client_Server Code structure for class TStack

By default there is only one instance of a *server_class and <class>_server*. which are located within the TStack Server space.

A4 ODS CONTENTS ILLUSTRATION

A4.1 ODS CONTENT IN STATE "CHILD"

Textual Description of the object
Signature of provided operations
associated Textual Description
Declaration of provided Types and Exceptions
associated Textual Description
Description of required interface
Textual Description of execution constraints and OBCS

A4.2 ODS CONTENT IN STATE "PARENT"

Textual Description of the object
Signature of provided operations
associated Textual Description
Declaration of provided Types and Exceptions
associated Textual Description
Description of required interface
Textual Description of execution constraints and OBCS
Declaration of child objects .i.enfant;s
Declaration of Type, Exception and Operration implementation

A4.3 ODS CONTENT IN STATE "TERMINAL"

Textual Description of the object
Signature of provided operations
associated Textual Description
Declaration of provided Types and Exceptions
associated Textual Description
Description of required interface
Textual Description of execution constraints and OBCS
Implementation of OBCS
Declaration and Implementation of internal Types
associated Textual Description
Declaration and Implementation of Data
associated Textual Description
Declaration of internal Opeérations
associated Textual Description
Impementationof operations (provided and internals)

A5 ABBREVIATION LIST

- ADT Abstract Data Type
- AM Abstract Machine
- APSE Ada Programming Support Environment
- ASER ASynchronous Execution Request
- ASCII American Standard Code for Interchange of Information
- BNF Backus Naur Form
- CDT **Class** Design Tree
- FSM Finite State Machine
- ER Execution Request
- FIFO First In First Out
- HDL HOOD Design Language
- HCS HOOD Chapter Skeleton
- HDT HOOD Design Tree
- HOOD Hierarchical Object Oriented Design
- HRM HOOD Reference Manual
- HRTS HOOD RUN TIME SUPPORT or Virtual Machine
- HSER Highly Synchronous Execution Request
- HTG HOOD Technical Group
- HUM HOOD User Manual
- HUG HOOD User Group
- HW Hardware
- HWG HOOD Working Group
- ISR Interrupt Service Routine
- LSER Loosely Synchronous Execution Request
- MTEX Mutual Exclusion constraint
- OBCS OBject Control Structure
- ODS Object Description Skeleton
- OOD Object Oriented Design
- OPCS OPeration Control Structure
- OR Observation Report
- OS Operating System

- OSTD Object State Transition Diagram
- OSTM Object State Transition Machine
- PDL Program Design Language
- RASER Reporting Asynchronous Execution Request
- RLSER Reporting Loosely Synchronous Execution Request
- SIF Standard Interchange Format
- STD State Transition Diagram
- TOER Timed Out Execution Request
- VN Virtual Node
- VNT Virtual Node Tree

[