Stood 4.1

User's Manual

part I: Project and Session management

Pierre Dissaux

1. Getting started

This section explains how to start a new user session with **STOOD**. It is supposed here that a standard installation procedure for the product and its license server has been followed successfully before attempting to use **STOOD**. Please refer to *Installation Manual* in case of any problem. **STOOD v4.1** is available for **Unix/Motif** and **Windows** platforms. A high level of interoperability is available between these two versions of the product.

1.1. Administrator's guide

This chapter contains usefull information to check current installation of the product on your system. Following components should be found after a standard installation of **STOOD v4.1** on your system:

1.1.1. Binary files

1.1.1.1. Supported platforms

`bin.xxx` directory contains all required platform specific binary files, where `xxx` identifies actual environment among the following:

- `hp700` for **Hp-ux** on **hp9000/700** platforms.
- `ibm` for **Aix** on IBM **RS6000** platforms.
- `sol2` for **Solaris2** on Sun **sparc** platforms.
- `w32` for **Windows** on **PC** platforms.
- `pclinux` for **Linux 2.2** on **PC** platforms.

Binaries for other platforms may be available. Please contact **TNI**'s technical support for further informations: `stood@tni.fr`
Please note that **VMS** on **Digital Vax** and **Alpha** is no more supported.

1.1.1.2. Executable files

Available executable files for a given platform are listed below. On Windows platforms, all executable files have a `.exe` extension:

`stood`	main executable to launch STOOD
`sbprolog`	prolog engine for post-processors
`scan_ada`	Ada lexical analyser
`scan_c`	C lexical analyser
`scan_cpp`	C++ lexical analyser
`scan_pseudo`	pseudo-code lexical analyser
`adarev`	Ada syntactic analyser

Note that best way to launch **STOOD** is not for the user to execute `stood` binary file directly inside `bin` directory. It is preferable to add `bin` directory to **Unix** execution path, and to launch **STOOD** from a user's owned working directory, or to create a shortcut for **Windows** platforms.

1.1.1.3. Ancillary files

A few ancillary files need to be located inside `bin` directory:

	Unix	Windows	
`stood.eng`	x	x	STOOD localization file
`stood.uid`	x		Motif ressources file
`startup.xpm`	x		startup picture
`win_nt`		x	protection key directory
`nslms32.dll`		x	protection key utility

1.1.1.4. Initialization file

`bin` directory also contains a default initialization file where customizable options and parameters may be set to fit user's preferences:

`.stoodrc`	default initialization file for Unix
`stood.ini`	default initialization file for Windows

Other copies of these files may be created and customized inside users working directories in order to manage several concurrent configurations. More details about initialization files contents and customization is provided in § 1.2.

If no other initialization file is found, **STOOD** will use the one located inside `bin` directory. Initialization file gathers all user's level customizations. Many other customization capabilities are available at administrator's level. These other customizable features are located inside `config` directory.

1.1.2. Configuration files

`config` directory is the general container for all platform independant configuration files, including documentation and code generators. Features contained in this directory may all be customized by a tool administrator.

Standard configuration complies as far as possible **HOOD** Reference Manual (**HRM**) release 4.0, September 1995, and has been extended thanks to numerous feedbacks from operational users and projects. More recently, support of Hard Real Time extensions (**HRT-HOOD**) as been added to **STOOD**.

Several configuration directories may be defined in order to fit specific requirements for a given **Project**. It is possible, for instance, to:

- define and implement particular code generation, documentation and verification rules;
- implement interacting utilities with other tools;
- customize help files;

To switch from a given configuration directory to another, `ConfigPath` property should be properly set within relevant initialization file (`stood.ini` or `.stoodrc`). Refer to §1.2.6 for further details.

1.1.2.1. Code generators

Code generators are located inside `code_extractors` configuration subdirectory. There is a dedicated subdirectory for each installed code generator:

- `config/code_extractors/ada` **Ada**
- `config/code_extractors/c` **C**
- `config/code_extractors/cpp` **C++**

Each of these subdirectories contains a set of files that are used by **STOOD** each time corresponding code extraction action is required. Code extractors are written in **prolog** language. When starting code extraction, **STOOD** produces a **prolog** facts base and gives the control to a **prolog** engine which loads both facts and rules bases, to generate code files (refer to § 1.1.4).

Contents of a code extractor directory is as follow. Some of these files may be customized by tool administrator.

`Extract.pro`	prolog rules (source code)
`Extract.sbp`	prolog rules (binary code)
`Init.pro`	prolog run-time interface (source)
`Init.sbp`	prolog run-time interface (binary)
`Input.sbp`	input file for code extraction (rules base)
`go.sh`	launching shell script
`scan.lex`	lexical analyser (lex code)
`scan.c`	lexical analyser (C code)
`special`	definition of code-dependent symbol types
`extractors`	definition of code extraction modes
`pragma`	definition of code extraction options
`makefile`	to re-build code extractor if required

More details about contents and use of these files is provided in part IV of this documentation. Like other the post-processors, code extractors may be updated more frequently than **STOOD** kernel. To know precise version of a code extractor, edit `Extract.pro` file, which header provides the date of last modifications.

1.1.2.2. Document generators

Documentation generators are located inside `doc_extractors` configuration subdirectory. Documentation may be produced in various format. There is a dedicated subdirectory for each installed document generator:

- `config/doc_extractors/html_p` **HTML** file
- `config/doc_extractors/tps_p` **Interleaf** input file
- `config/doc_extractors/mif_p` **FrameMaker** input file
- `config/doc_extractors/ps_p` **PostScript** file
- `config/doc_extractors/rtf` **MSWord** input file
- `config/doc_extractors/latex` **LaTeX** file

Each of these subdirectories contains a set of files that are used by **STOOD** each time corresponding document creation is requested.

Document generators are written either in **prolog** language (those which name ends by `_p`), either using a specific scripting language called **easyDoc**. Both kinds of generators may be customized by tool administrator. Following files should appear in each document generator:

`variable.cfg`	definition of document variables
`suffix.cfg`	definition of output file suffix
`keepps.cfg`	specifies to keep temporary EPSF files (optional)

In addition to `.cfg` files, **prolog** document generator directories contain:

`Extract.pro`	prolog rules (source code)
`Extract.sbp`	prolog rules (binary code)
`Init.pro`	prolog run-time interface (source)
`Init.sbp`	prolog run-time interface (binary)
`Input.sbp`	input file for code extraction (rules base)
`print.sh`	main launching shell script
`printer.sh`	additional script to send to a printer
`preview.sh`	additional script to send to a pre-viewer
`header.xxx`	initializations, tags definitions
`prolog`	identifies a prolog generator
`makefile`	to re-build doc generator if required

Important notes:

- File `printer.sh` is used to send produced document to a printer or a documentation tool. Tool administrator should customize there the actual name of used printer or print spooler.
- File `header.xxx` (where `xxx` may be `ps`, `tps` or `mif`), may be edited to customize documentation fonts.
- Other `.sh` files may be created to propose different printing modes or different printers to the user. When only `print.sh` is defined, only *file only* menu option is proposed in *document editor*. When additional scripts are defined, corresponding entries are automatically proposed in *document editor*.
- **STOOD** *document editors* are described in §5 of part III of the documentation

In addition to `.cfg` files, **easyDoc** document generator directories contain a list of files describing a sequence of instructions to be inserted at the beginning and at the end of document sections:

`doc`	`doc_`	begin and end of the document
`par`	`par_`	begin and end of a paragraph
`sect#`	`sect#_`	begin and end of a section (#: 0..7)
`bold`	`bold_`	begin and end of bold text
`italic`	`italic_`	begin and end of italic text
`fixed`	`fixed_`	begin and end of fixed font text
`verb`	`verb_`	begin and end of formated text
`epsf`		insertion of an EPSF file

Other specialized documentation procedures may be defined. To add a new documentation procedure, is required to first modify `DataBase` configuration file to define a new `DocProc` tag (refer to § 1.1.2.7). Then, a pair of files named `newtag` and `newtag_` (where `newtag` represents the actual name of the new documentation tag) must be created within relevant **easyDoc** documentation format directory. These files must contain appropriate processing for entering and exiting a documentation section.

1.1.2.3. Rules checkers

Rules checkers are located inside `checkers` configuration subdirectory. There is a dedicated subdirectory for each installed code rules checker:

- `config/checkers/hood` **HOOD** v4 rules checker
- `config/checkers/metric` design metrics

Each of these subdirectories contains a set of files that are used by **STOOD** each time corresponding verification action is required. Rules checkers are written in **prolog** language. When starting code extraction, **STOOD** produces a **prolog** facts base and gives the control to a **prolog** engine which loads both facts and rules bases, to generate check reports.

Contents of a rules checker directory is as follow. Some of these files may be customized by tool administrator.

`_Main.pro`	prolog main rule (source code)
`_Main.sbp`	prolog main rule (binary code)
`_Init.pro`	prolog run-time interface (source)
`_Init.sbp`	prolog run-time interface (binary)
`_Input.sbp`	input file for code extraction (rules base)
`go.sh`	launching shell script
`makefile`	to re-build rules checker if required

In addition to these common files, hood checker directory contains a pair of files for each category of rules to be checked. These files are:

`General.pro (.sbp)`	general HOOD rules
`Include.pro (.sbp)`	rules for Include relationships
`Use.pro (.sbp)`	rules for Use relationships
`Operation.pro (.sbp)`	rules for Operations
`Provided.pro (.sbp)`	rules for Provided Interfaces
`Visibility.pro (.sbp)`	visibility rules
`Consistency.pro (.sbp)`	consistency rules
`Required.pro (.sbp)`	rules for Required Interfaces
`Std.pro (.sbp)`	additional rules for States &Transitions

More details about contents and use of these files is provided in part IV of this documentation. Like other the post-processors, rules checkers may be updated more frequently than **STOOD** kernel.

1.1.2.4. Tools

STOOD uses **Unix** shell scripts to control the interface between the kernel and post-processors or file storage environment, and to easily call remote tools. These scripts may all be customized by tool administrator, if required, but for safety reasons, they are stored into two different configuration subdirectories: `internalTools` and `externalTools`.

Internal tools should never be removed as they are the gateway between the kernel and post-processors (rules checkers, code and document generators) and file system. Contents of `config/internalTools` configuration subdirectory is:

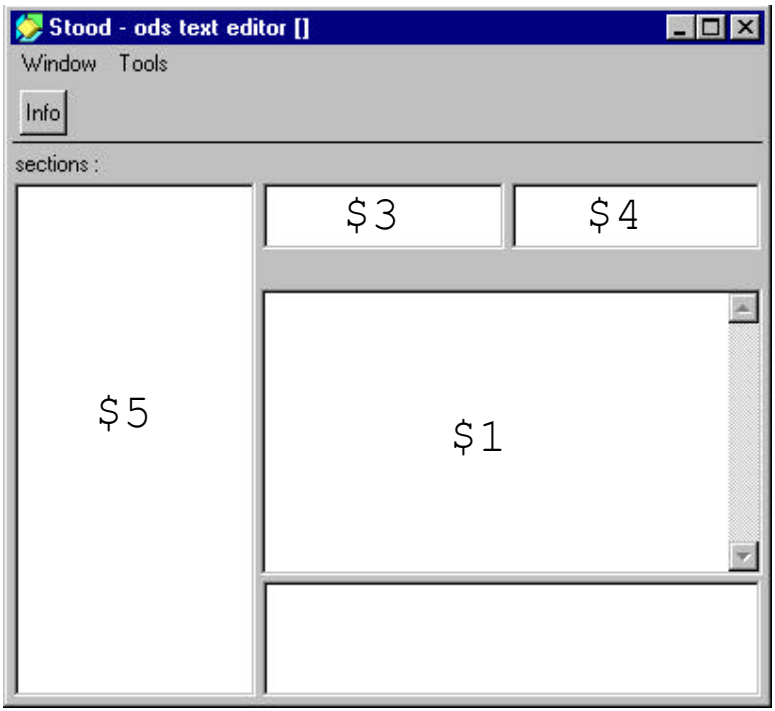
`lock.sh`	called when openning an Application
`inittrash.sh`	called when closing an Application
`infosyc.sh`	called when inquiring about a Project
`inforoot.sh`	called when inquiring about an Application
`copydir.sh`	called when copying or moving files
`rmdir.sh`	called when deleting files
`fastprint.sh`	called to print graphics and trees directly
`print.sh`	called to print compound documents
`scan.sh`	called when accepting source code
`external.sh`	called to launch checkers and extractors
`difffiles.sh`	called when comparing files

On the contrary, external tools are not mandatory. They may be defined to communicate with remote tools. They can be called only from *text editors*. Default contents of `config/externalTools` configuration subdirectory is described below. This contents should be considered as an example only.

`info.sh`	provide information about current selection
`emacs.sh`	launch emacs editor (if possible)
`lpr.sh`	send selected file contents to a printer
`check_ada.sh`	launch gnat for Ada code analysis (if possible)
`make.sh`	execute selected makefile

It is possible to send information to external tools via five parameters which value is related to current selected items in used *text editor*.

$1	file pathname for storage area of current section
$2	current Application name
$3	current Module name
$4	current Component name
$5	current section identifier (logical name)



Result of an external tool execution is displayed in a dialog box, which contains information sent to shell script standard output. Note that execution of an external tools suspends STOOD until its completion.

1.1.2.5. Contextual help files

A on-line help mecanism is available with **STOOD**. It is also fully customizable by tool administrator. Help facility is composed of three different parts, each of them stored in a dedicated configuration subdirectory:

- `config/help` help files for **STOOD** windows
- `config/ods_help` help files for **ODS** sections
- `config/ods_template` templates for **ODS** sections

Contents of `help` subdirectory is a list of files, attached to each editor or dialog box. Help may be provided at two levels.

Information contained in these files is displayed in a dialog box when corresponding *help* menu or button has been selected. A *more help* button gives access to more detailed information, if any. An additional file may be created in each case to provide this second level help. These additional files should have a `.more` suffix.

Help filenames are directly related to window identifiers also used for setting initialization file properties (refer to §1.2.4) or in **STShell** language (refer to §1.3.1.1).

`main`	main editor
`syc`	system editor
`gra`	graphical editor
`gra_txt_none`	text input area of graphic editor (no selection)
`gra_txt_obj`	text input area of graphic editor (module selected)
`gra_txt_ope`	text input area of graphic editor (operation selected)
`gra_txt_typ`	text input area of graphic editor (type selected)
`gra_txt_con`	text input area of graphic editor (constant selected)
`gra_txt_exc`	text input area of graphic editor (exception selected)
`gra_txt_dat`	text input area of graphic editor (data selected)
`vna`	allocation editor
`hie`	inheritance tree
`std`	states-transitions diagram editor
`txt`	text editor
`crf`	cross-references table
`chk`	any rule checker
`utr`	call tree
`ext`	code extractor
`rev`	code reversor
`doc`	document editor
`sch`	documentation scheme editor
`dbcfg`	options dialog box
`dbobj`	module selection dialog box
`dbobjla`	module and language selection dialog box
`dbcompare`	designs comparison dialog box
`dbcopy`	design copy dialog box
`dbreplace`	design replace dialog box

Specific help is provided for editing text editors sections. This is particularly useful to give user advices about the way to insert information in **ODS** sections. These advices may be informative text or examples of text input that are directly inserted at the right place.

Both may be customized by tool administrator, by editing files contained in `ods_help` and `ods_template` configuration subdirectories.

Organization of these two subdirectories is related to the way **Application** storage have been configured (refer to §1.1.2.7). Help and template information files are organized as any **STOOD Application**, but in a generic way. Each time an **Application**, **Module** or **Component** name is required to build actual storage pathname (refer to §2.2.2.2), reserved keyword `name` is used instead.

It is also possible to provide information for sections that are not stored in a file, but deduced from graphics by a procedure. In this case, help and template files will be named `proc#`, where `#` is the procedure number defined inside `DataBase` file. Many sections controled by procedure are read-only, so that only help information is provided (no template).

These help and template files may use following contextual pseudo-variable:

`$Dg`	Application name	`$Id`	RCS tag
`$Op`	Operation name	`$S1`	Super-Class (Ada syntax)
`$Os`	Operation-Set name	`$S2`	Super-Class (C++ syntax)
`$Ty`	Type name	`$A1`	Attributes (Ada syntax)
`$Cp`	Constant name	`$A2`	Attributes (C++ syntax)
`$Ex`	Exception name	`$Ho`	current SavePath directory
`$Da`	Data name	`$St`	current config directory
`$Se`	State or Transition name	`$Ts`	Test sequence name

Full contents of `ods_help` and `ods_template` configuration subdirectories would be at first directory level (sections global to an **Application**):

- `name` directory for second directory level (**Modules**)
- `proc#` information file for procedure `#` (see table below)

`proc1`	list of child Modules	*Read Only*
`proc2`	contents of current System Configuration	*Read Only*
`proc4`	DataFlows	*Read Only*
`proc5`	Exception Flows	*Read Only*
`proc15`	actual parameters for Instance_Of Generic Modules	
`proc16`	instance range for Instance_Of Generic Modules	
`proc22`	Operation declaration	
`proc23`	Used Operations	*Read Only*
`proc30`	Class inheritance	
`proc31`	Class attributes	
`proc32`	Exception definition	
`proc33`	propagated Exceptions	
`proc34`	Constrained Operations	
`proc35`	OBCS is Implemented By	*Read Only*
`proc36`	Required Interface	*Read Only*
`proc37`	Operation Set definition	*Read Only*
`proc61`	Operation is Implemented By	*Read Only*
`proc62`	Type is Implemented By	*Read Only*
`proc63`	Constant is Implemented By	*Read Only*
`proc64`	Exception is Implemented By	*Read Only*
`proc65`	Data is Implemented By (forbidden)	*Read Only*
`proc66`	Operation Set is Implemented By	*Read Only*
`proc81`	Operation Set contents	*Read Only*
`proc91`	symbol is used by	*Read Only*
`proc93`	symbol uses	*Read Only*
`proc224`	Transition event	*Read Only*
`proc225`	State exiting Transitions	*Read Only*
`proc226`	State entering Transitions	*Read Only*
`proc227`	Transition origin State	*Read Only*
`proc228`	Transition destination State	*Read Only*

At second directory level (sections global to a **Module**), `ods_help/name` and `ods_template/name` contain a set of files and a set of directories:

- files:

`PRAGMA`	help and template files for Module Pragmas
`specHeader.u`	help and template files for Ada spec file header
`specHeader.c`	help and template files for C spec file header
`specHeader.cc`	help and template files for C++ spec file header
`bodyHeader.u`	help and template files for Ada body file header
`bodyHeader.c`	help and template files for C body file header
`bodyHeader.cc`	help and template files for C++ body file header
`modif`	help and template files for Module changes file

- `DOC` subdirectory (help and template for **Description** files):

`StaPro.t`	help and template files for Statement of the Problem
`RefDoc.t`	help and template files for Referenced Documents
`StrReq.t`	help and template files for Structural Requirements
`FunReq.t`	help and template files for Functional Requirements
`BehReq.t`	help and template files for Behavioural Requirements
`ParDes.t`	help and template files for Parent Description
`UseMan.t`	help and template files for User Manual Outline
`GenStr.t`	help and template files for General Strategy
`IdeChi.t`	help and template files for Identification of Children
`IdeStr.t`	help and template files for Identification of Types
`IdeOpe.t`	help and template files for Identification of Operations
`GroOpe.t`	help and template files for Grouping Operations
`IdeBeh.t`	help and template files for Identification of Behaviour
`JusDes.t`	help and template files for Justification of Decisions
`ImpCon.t`	help and template files for Implementation Constraints
`header`	help and template files for code files header

Additional sections for **HRT-HOOD** Real-Time attributes of a **Module**:

`CeiPri.hrt`	help and template files for Ceiling Priority
`Period.hrt`	help and template files for Period
`Offset.hrt`	help and template files for Offset
`MinTim.hrt`	help and template files for Minimum Arrival Time
`MaxFreq.hrt`	help and template files for Maximum Frequency
`Ddline.hrt`	help and template files for Deadline
`Priori.hrt`	help and template files for Priority
`PreCon.hrt`	help and template files for Precedence Constraints
`TimTra.hrt`	help and template files for Time Transformation
`Import.hrt`	help and template files for Importance

- `OP` subdirectory (help and template for **Operations**):

`name.t`	help and template files for Operation spec description
`name.t2`	help and template files for Operation body description
`name.hx`	help and template files for Operation handled Exceptio
`name.x`	help and template files for Operation Ada extension
`name.p`	help and template files for Operation Pseudo code
`name.u`	help and template files for Operation Ada code
`name.c`	help and template files for Operation C code
`name.cc`	help and template files for Operation C++ code
`name_test.t`	help and template files for Operation test description
`name_prec.t`	help and template for Op. preconditions description
`name_prec.u`	help and template for Op. preconditions Ada code
`name_post.t`	help and template for Op. postconditions description
`name_post.u`	help and template for Op. postconditions Ada code
`name_modif`	help and template for Operation changes file
`name_header.u`	help and template for Op. Ada separate file header

Additional sections for **HRT-HOOD** Real-Time attributes of an **Operation**:

`name_budg.hrt`	help and template files for Operation budget
`name_wcet.hrt`	help and template files for Operation WCET

- `T` subdirectory (help and template for **Types**):

`name.t`	help and template files for Type textual description
`name.s`	help and template files for Type Ada pre-declaration
`name.u`	help and template files for Type Ada full definition
`name.h`	help and template files for Type C definition
`name.hh`	help and template files for Type C++ definition

- `C` subdirectory (help and template for **Constants**):

`name.t`	help and template files for Constant textual description
`name.s`	help and template files for Constant Ada pre-declaration
`name.u`	help and template files for Constant Ada full definition
`name.h`	help and template files for Constant C definition
`name.hh`	help and template files for Constant C++ definition

- `D` subdirectory (help and template for **Data**):

`name.t`	help and template files for Data textual description
`name.s`	help and template files for Data Ada definition
`name.c`	help and template files for Data C definition
`name.cc`	help and template files for Data C++ definition

- `STD` subdirectory (help and template for **States** and **Transitions**):

`obcs.t`	help and template files for OBCS spec description
`obcs.t2`	help and template files for OBCS body description
`obcs.p`	help and template files for OBCS Pseudo code
`obcs.u`	help and template files for OBCS Ada code
`obcs.c`	help and template files for OBCS C code
`obcs.cc`	help and template files for OBCS C++ code
`name.t`	help and template files for State textual description
`name_set.u`	help and template files for State assignment in Ada
`name_get.u`	help and template files for State test code in Ada
`name_set.c`	help and template files for State assignment in C
`name_get.c`	help and template files for State test code in C
`name_set.cc`	help and template files for State assignment in C++
`name_get.cc`	help and template files for State test code in C++
`name.t2`	help and template files for Transition description
`name_cnd.u`	help and template files for Transition condition in Ada
`name_exc.u`	help and template files for Transition exception in Ada
`name_cnd.c`	help and template files for Transition condition in C
`name_exc.c`	help and template files for Transition exception in C
`name_cnd.cc`	help and template files for Transition condition in C++
`name_exc.cc`	help and template files for Transition exception in C++
`OBCS_header.u`	help and template files for OBCS Ada sep. file header

- `X` subdirectory (help and template for **Exceptions**):

`name.t`	help and template files for Exception description

- `OPS` subdirectory (help and template for **Operation Sets**):

`name.t`	help and template files for Operation Set description

- `OTS` subdirectory (help and template for Test Sequence files):

`name_desc.t`	help and template files for Test sequence description
`name_sequ.u`	help and template files for Test Ada code

1.1.2.6. Icons

STOOD uses a few customizable icons, especially when displaying buttons or menu items. Icons definition files are stored in `icons` configuration subdirectory. Each icon is described in `.bmp` format for **Windows** platforms, and in `.xpm` format for **Unix** platforms. Default contents of `config/icons` subdirectory is:

`aggreg.bmp`	`aggreg.xpm`
`class.bmp`	`class.xpm`
`compon.bmp`	`compon.xpm`
`connect.bmp`	`connect.xpm`
`datafl0.bmp`	`datafl0.xpm`
`datafl1.bmp`	`datafl1.xpm`
`datafl2.bmp`	`datafl2.xpm`
`delete.bmp`	`delete.xpm`
`except.bmp`	`except.xpm`
`inherit.bmp`	`inherit.xpm`
`istate.bmp`	`istate.xpm`
`object.bmp`	`object.xpm`
`spark.bmp`	`spark.xpm`
`state.bmp`	`state.xpm`

It is possible to edit these files with appropriate utility program to change the icons, or add other icons and associate them to window buttons (refer to § 1.2.4.1).

1.1.2.7. DataBase file

The place where **Application** data storage is defined is `config/DataBase` description file. It may be required to customize this file to perform following king of changes:

- Add or remove sections in standard **HOOD ODS**.
- Add sections for new target languages (**Fortran**, **Java**, ...)
- Create or customize textual editors
- Change documentation framework
- Modify **Application** storage organization
- ...

Contents of this file consists in a sequential list of records, one for each section of any text editor. These records should comply with a precise syntax which is described below with a simple variant of Backus-Naur Form (BNF) where:

- Plain words are used to denote syntactic rules identifiers
- Boldface words are used to denote keywords
- Square brackets enclose optional items
- Curly brackets enclose a repeated item
- A vertical line separates alternative items

```
(1)DataBase ::= { Section[2] }

(2)Section ::= Label[3] LogicalName[4] (
  SectionLevel[5] [ModuleMask[6]]
  [SectionStorage[7]] [Title[8]] [LoopProc[9]]
  DocProc[10] [EditorMask[11]] [ChildPropagate] )
```

```
(3)Label ::= string
```

`Label` is the string that is visible in section area of text editors. This string value may be customized without any constraint.

```
(4)LogicalName ::= identifier
```

On the contrary, `LogicalName` should not be modified as it may be used by **STOOD** as an internal identifier.

```
(5)SectionLevel ::= level positive
```

`SectionLevel` is used to manage section hierarchy. It is used to indent labels in text editors, and to define a hierarchy of paragraphs in produced documentation. Highest level is 1, and in standard configuration, lowest level is 6. Note that `ModuleMask` is automatically inherited from higher level sections.

```
(6)ModuleMask ::= when BooleanExpression[12]

(12)BooleanExpression ::= ModuleKind[13]
  { BooleanOperator[14] ModuleKind[13] }

(13)ModuleKind ::= a | o | i | f | e | c
| sroot | root2 | root | leaf | constr | sif

(14)BooleanOperator ::= + | . | \
```

The way **STOOD** knows if a section is relevant for a given kind of **Modules**, is value of `ModuleMask` expression. Meaning of `ModuleKind` constants is:

a	Active Module
o	Op_Control Module
i	Instance_Of Module
f	Formal_Parameters Module
e	unbound Environment Module
c	Class Module
sroot	System_Configuration
root2	bound Environment Module
root	Root_Module
leaf	Terminal Module
constr	Module providing at least one Constrained Operation
sif	to specify that this section should not appear in SIF files

Additional **Module** kinds has been defined to support **HRT-HOOD**. Please note that those ones are not hardwired like the others, but are defined within initialization file (`stood.ini` or `.stoodrc`). Refer to § 1.2.8.

cy	HRT Cyclic Object
sp	HRT Sporadic Object
pr	HRT Protected Object

These cases may be combined using boolean operators:

+	logical OR
.	logical AND
\	logical NOT

```
(7)SectionStorage ::=
  text pathname
| text procNumber
| dir pathname
```

The way **STOOD** knows how to get or store information related to this section, is specified by `SectionStorage`. Provided parameter may be either a file pathname, either an internal procedure number.

Each `Pathname` is specified in a generic way, using a **Unix** syntax (even for **DOS** based platforms) and following pseudo-variables:

`$Ho`	pathname of current storage directory (SavePath)
`$St`	pathname of configuration directory (ConfigPath)
`$Dg`	name of current Application
`$Ob`	name of current Module
`$Op`	name of current Operation (if relevant)
`$Tp`	name of current Type (if relevant)
`$Cp`	name of current Constant (if relevant)
`$Os`	name of current Operation Set (if relevant)
`$Ex`	name of current Exception (if relevant)
`$Da`	name of current Data element (if relevant)
`$Se`	name of current State or Transition (if relevant)
`$Ts`	name of current test sequence (if relevant)

When information is produced by an internal procedure, `procNumber` should be one of the following:

1	list of child Modules	61	Operation is Implemented By
2	contents of System Config.	62	Type is Implemented By
4	DataFlows	63	Constant is Implemented By
5	Exception Flows	64	Exception is Implemented By
15	parameters for Instance_Of	65	Data is Implemented By
16	instance range for Instance_Of	66	Operation Set Implemented By
17	begin of ODS	81	Operation Set contents
18	type of current Module	91	symbol is used by
19	end of ODS	92	symbol name
20	OPCS begin	93	symbol uses
21	OPCS end	94	Call Tree
22	Operation declaration	95	Inverse Call tree
23	Used Operations	199	Inheritance Tree
30	Class inheritance	200	Design Tree
31	Class attributes	201	Operations Diagram
32	Exception definition	202	Types Diagram
33	propagated Exceptions	203	Constants Diagram
34	Constrained Operations	204	Exceptions Diagram
35	OBCS is Implemented By	205	Data Diagram
36	Required Interface	211	Parent Operations Diagram
37	Operation Set definition	212	Parent Types Diagram
41	child Operation	213	Parent Constants Diagram
42	child Type	214	Parent Exception Diagram
43	child Constant	215	Parent Data Diagram
44	child Exception	220	STD
45	child Data	221	Parent STD
51	Operation name	222	State name
52	Operation Set name	223	Transition name
53	Type name	224	Transition event
54	Constant name	225	State exiting Transitions
55	Exception name	226	State entering Transitions
56	Data name	227	Transition origin State
		228	Transition destination State

```
(8)Title ::=
  title string
| title procNumber
| title nil
```

Is is possible to control the string that will be used for section title in printed documents. If `Title` field is missing, then `SectionLabel` will be used to print section title. If a string constant is given, then it will be used as a title. If a proper procedure number is provided, then **STOOD** will generate dynamically title to be printed. Finally, if `nil` keyword is specified, then no title will be printed.

```
(9)LoopProc ::= list LoopNumber[15]

(15)LoopNumber ::= 90 | 92 | 95 | 96 | 1X[16]Y[17]Z[18]

(16)X ::= 1 | 2 | 3 | 4 | 5
(17)Y ::= 1 | 2
(18)Z ::= 0 | 1 | 2
```

Some `DataBase` file sections are related to a unique entity, but to a list of entities of the same kind. This is the case when a **Component** is selected in a text editor. `LoopNumber` field is used to specify which list processing is required. Encoding is as follow:

`90`	list of rules checker categories
`92`	list of cross-references table symbols
`95`	list of States
`96`	list of Transitions
`1XYZ`	list of Operations, Types, Constants, Exceptions and Data

In last case, X, Y and Z digits may have following values:

X	
1	list of Operations
2	list of Types
3	list of Constants
4	list of Exceptions
5	list of Data

Y	
1	element
2	set

Z	
1	Provided
2	Internal
3	both

(10)DocProc ::= **doc** DocType[19]

(19)DocType ::= **TXT** | **CODE** | **TXTEND** | **POSTSCRIPT**

A specific documentation procedure may be applied to a section. These procedures must be implemented in each document generator. Standard procedures are:

TEXT	plain text
CODE	fixed font text
TEXTEND	plain text without form feed
POSTSCRIPT	Encapsulated PostScript File insertion

```
(11)EditorMask ::= flags BooleanExpression2[20]

(20)BooleanExpression2 ::=
  EditorId[21] { BooleanOperator[14] EditorId[21] }

(21)EditorId ::=
  eOds | eAda | eC | eCpp | eChecks | eTests
```

With `EditorMask` section field, it is possible to specify in which text editor this section will be visible. This field may also be used to create new customized text editors in **STOOD**. Standard text editors are:

`eOds`	ods text editor
`eAda`	ada text editor
`eC`	c text editor
`eCpp`	cpp text editor
`eChecks`	checks text editor
`eTests`	tests text editor

To create a new text editor, first referencing section must declare it in its EditorMask field:

```
flags (eNew='my_editor')
```

In this case, a *my_editor text editor* will be automatically added to standard text editors.

Finally, **`ChildPropagate`** field provides a way to make information be propagated along **Implemented_By** links. If this field is present, then a section of a **Non Terminal Module** will point to the contents of regarding section in relevant **Terminal Module**, if **Implemented_By** relationship have been properly set.

Example:

```
'operation spec. description (text)'   OpTxt
    (level 5  when \root2+f       list 1110
     text '$Ho/$Dg/$Ob/OP/$Op.t'
     doc TXT  flags eOds)

'operation declaration (hood)'         OpDecl
    (level 5                      list 1110
     text 22
     doc CODE flags eOds + eAda + eC + eCpp)
```

First section contains informal text stored in a file. It concerns all the **Provided Operations** of any **Module**, except bounded **Environments**. It will be visible only in *ods text editor*.

Second section contains code calculated by an internal procedure. It concerns also **Provided Operations** of any **Module**. It is visible in *ods text editor*, *ada text editor*, *c text editor* and *cpp text editor*.

1.1.3. Applications examples

STOOD standard installation contains a set of directories with **Application** examples that may differ from a delivery to another, and typically:

- `exAda95`: examples implemented in **Ada95**
- `exCpp`: examples implemented in **C++**
- `exC`: examples implemented in **C**
- `exMacros`: examples of **STShell** macro commands
- `exHRT`: examples for **HRT-HOOD**
- `libs`: interfaces to standard libraries (**Ada**, **C**, **C++**)

To get access to examples, these directories must be listed in `SavePath` property of initialization file (refer to §1.2.2).

1.1.4. Prolog engine

1.1.4.1. sbprolog

`sbprolog` directory, contains sources and librairies of the **prolog** environment developed by the **S**tate **U**niversity of **N**ew **Y**ork at Stony Brook (`http://www.sunysb.edu/`). If no other **prolog** engine is available, **sbprolog** will be used to perform post-processing actions (code extraction, rules checking, document generation).

STOOD post-processors **prolog** source code is provided in standard delivery in order to let tool administrator use another **prolog** environment, if needed.

STOOD doesn't require source files of **prolog** engine and libraries to work properly. They may thus be removed from **STOOD** execution environment. Minimum contents of `sbprolog` directory should nevertheless be:

`lib`	sbprolog library
`modlib`	sbprolog library
`cmplib`	sbprolog library
`prolog`	shell script to launch prolog interpreter
`compile`	shell script to re-build STOOD post-processors

Executable file for **prolog** engine is located into `bin.xxx` directory. **STOOD** always launches **prolog** engine via **Unix** shell scripts:

`checkers/*/go.sh`	rules checkers
`code_extractors/*/go.sh`	code extractors
`doc_extractors/*_p/print.sh`	document generator

Each script contains at least statements similar to the following:
Access path to **sbprolog** libraries:

```
SIMPATH=modlib:lib:cmplib; export SIMPATH
```

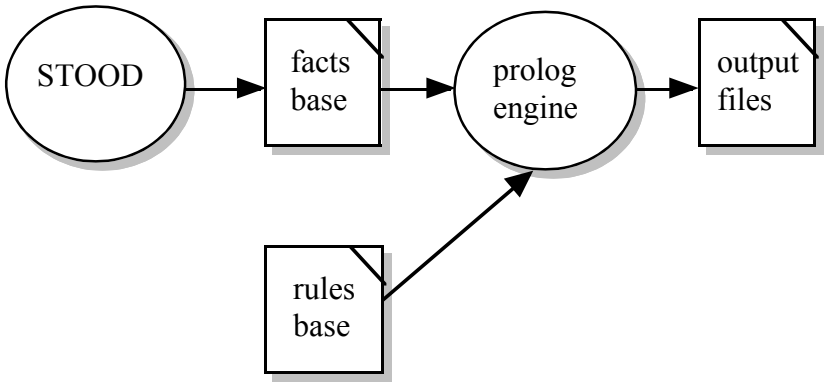
Launching sbprolog executable file:

```
sbprolog -m 500000 -p 500000 Input.sbp
```

`STOODBIN` and `STOODPRO` environment variables are used to provide actual location of `bin.xxx` and `sbprolog` directories (refer to § 1.2.6).

1.1.4.2. prolog interface

STOOD communicates with the **prolog** engine within a dedicated file interface. Post-processors consists in a set of **prolog** rules, whereas **STOOD** provides a set of facts, or predicates, describing current status of the **Application**, and options for the action to be performed.



Facts base file is dynamically generated into relevant output directory withing current **Application** storage area, before launching **prolog** engine:

`_checks/extract.pro`	rules checkers
`_ada/extract.pro`	Ada code extractor
`_c/extract.pro`	C code extractor
`_cpp/extract.pro`	C++ code extractor
`_doc/extract.pro`	document generators

The list of generated **prolog** predicates is:

- `isRootObject(Root,Kind,Path).`

`Root`	name of a Root Module in current system
`Kind`	DESIGN, GENERIC or VIRTUAL_NODE
`Path`	actual pathname of regarding storage area

- `isCurrentRoot(Root).`

`Root`	name of current Root Module

- `isMissing(Root).`

`Root`	name of a Root for which details are missing

- `isObject(Module,Kind,Parent).`

`Module`	name of a Module in current hierarchy
`Kind`	PASSIVE, ACTIVE, OP_CONTROL, ...
`Parent`	name of parent Module in current hierarchy

- `objectLevel(Module,Level).`

`Module`	name of a Module in current hierarchy
`Level`	depth in the hierarchy, 1 for the Root Module

- `isProvided(Component,Kind,Module).`

`Component`	name of a Provided Component in specified Modul
`Kind`	OPERATION, TYPE, CONSTANT, EXCEPTION
`Module`	name of the Module

- `isInternal(Component,Kind,Module).`

`Component`	name of an Internal Component in specified Modul
`Kind`	OPERATION, TYPE, CONSTANT, DATA, ...
`Module`	name of the Module

- `isImplementedBy(Pcomp,Kind,Pmod,Ccomp,Cmod).`

`Pcomp`	name of a Provided Component of Module Pmod
`Kind`	OPERATION, TYPE, CONSTANT, EXCEPTION,
`Pmod`	name of a Non Terminal Module
`Ccomp`	name of a Provided Component of Module Cmod
`Cmod`	name of a Child Module of Pmod

- `uses(Client,Server,View,Style).`

`Client`	name of a Module in current hierarchy
`Server`	name of another Module
`View`	OPERATION or TYPE
`Style`	1: Uses; 2: Inherits; 3: Attributes

- `argument(Op,'OPERATION',Mop,Mode,P,Mty,T,V,K).`

`Op`	name of an Operation of Module Mop
`Mop`	name of a Module of current hierarchy
`Mode`	in; out or in out
`P`	name of a Parameter of Operation Op
`Mty`	name of another Module
`T`	name of a Type of Module Mty
`V`	initial value for Parameter P
`K`	BY_VALUE; BY_POINTER; BY_REFERENCE

- `return(Op,'OPERATION',Mop,Mty,T,K).`

`Op`	name of an Operation of Module Mop
`Mop`	name of a Module of current hierarchy
`Mty`	name of another Module
`T`	name of a Type of Module Mty
`K`	BY_VALUE; BY_POINTER; BY_REFERENCE

- `isMemberOf(Op,'OPERATION',Module,Opset).`

`Op`	name of an Operation of Specified Module
`Module`	name of a Module of current hierarchy
`Opset`	name of an Operation Set of specified Module

- `isConstrained(Op,'OPERATION',Module,C,P).`

`Op`	name of an Operation of specified Module
`Module`	name of a Module of current hierarchy
`C`	STATE; HSER; LSER; ASER; BY_IT; TO; ROER
`P`	value of Constraint parameter, if any

- `raisedException(Op,'OPERATION',Module,Exc).`

`Op`	name of an Operation of specified Module
`Module`	name of a Module of current hierarchy
`Exc`	name of an Exception of specified Module

- `isInstance(Module,Instance,Generic).`

`Module`	name of an Instance Of Module in current hierarch
`Instance`	actual name of the instance (unused)
`Generic`	name of regarding Generic Module

- `formalParameter(Component,Kind,Generic).`

`Component`	name of a Formal Parameter of specified Generic
`Kind`	OPERATION, TYPE, CONSTANT
`Generic`	name of a Generic of current system

- `actualParameter(Comp,Kind,Instance,Value).`

`Comp`	name of a Formal Parameter of a Generic
`Kind`	OPERATION, TYPE, CONSTANT
`Instance`	name of an Instance Of Generic
`Value`	actual value for specified Parameter

- `isState(Module,State,Kind).`

`Module`	name of a Module of current hierarchy
`State`	name of a State of specified Module
`Kind`	1 for initial State, 0 otherwise

- `isTransition(Module,Transition,From,To,Event).`

`Module`	name of a Module of current hierarchy
`Transition`	name of a Transition of specified Module
`From`	name of origin State of specified Transition
`To`	name of destination State of specified Transition
`Event`	name of a Provided Operation of specified Module

- `isClass(Type,Module).`

`Type`	name of a Class of specified Module
`Module`	name of a Module of current hierarchy

- `isAbstract(Component,Kind,Module).`

`Component`	name of a Component of specified Module
`Kind`	TYPE or OPERATION
`Module`	name of Module of current hierarchy

- `isInherited(Operation,Module).`

`Operation`	name of an Operation of specified Module
`Module`	name of a Module of current hierarchy

- `inherits(Class,Mc,Superclass,Msc).`

`Class`	name of Class of Module Mc
`Mc`	name of a Module of current hierarchy
`Superclass`	name of a Class of Module Msc
`Msc`	name of another Module

- `attributes(Type,Mt,Attribute,Ta,Mta,Value).`

`Type`	name of a Type of Module Mt
`Mt`	name of a Module of current hierarchy
`Attribute`	name of an Attribute of specified Type
`Ta`	name of a Type of Module Mta
`Mta`	name of another Module
`Value`	default value for specified Attribute

- `requires(Ccomp,Ck,Cmod,Scomp,Sk,Smod,Ln).`

`Ccomp`	name of a Component of Module Cmod
`Ck`	OPERATION, TYPE, CONSTANT, EXCEPTION
`Cmod`	name of a Module of current hierarchy
`Scomp`	name of a HOOD Component of Module Smod
`Sk`	OPERATION, TYPE, CONSTANT, EXCEPTION
`Smod`	name of a Module (may be same as Cmod)
`Ln`	logical name of an ODS section

- `specialrequires(Ccp,Ck,Cmod,Ssymb,Sk,Smod,Ln).`

`Ccp`	name of a Component of Module Cmod
`Ck`	OPERATION, TYPE, CONSTANT, EXCEPTION
`Cmod`	name of a Module of current hierarchy
`Ssymb`	name of a symbol of Module Smod
`Sk`	parameter, attribute, temporary, enumeration, to be
`Smod`	name of a Module (may be same as Cmod)
`Ln`	logical name of an ODS section

Please note that an optional first parameter may be specified to identify references regarding ancillary languages.

- `description(Module,File,Ln).`

`Module`	name of a Module of current hierarchy
`File`	file pathname
`Ln`	logical name of an ODS Description section

- `comment(Component,Kind,Module,File,Ln).`

`Component`	name of a Component of specified Module
`Kind`	OPERATION, TYPE, CONSTANT, EXCEPTION,
`Module`	name of a Module of current hierarchy
`File`	file pathname
`Ln`	logical name of an ODS Txt section

- `file(Component,Kind,Module,File,Ln).`

`Component`	name of a Component of specified Module
`Kind`	OPERATION, TYPE, CONSTANT, EXCEPTION,
`Module`	name of a Module of current hierarchy
`File`	file pathname
`Ln`	logical name of an ODS default language section

- `file(Language,Component,Kind,Module,File,Ln).`

`Language`	name of a target language
`Component`	name of a Component of specified Module
`Kind`	OPERATION, TYPE, CONSTANT, EXCEPTION,
`Module`	name of a Module of current hierarchy
`File`	file pathname
`Ln`	logical name of an ODS specified language section

- `rcsId(Header).`

`Header`	value of configuration management tag

- `thisFile(Directory,File).`

`Directory`	directory containing current facts base file
`File`	current facts base file

- `fileProlog(Component,Kind,Module,File,Ln).`

`Component`	NIL
`Kind`	NIL
`Module`	name of a Module of current hierarchy
`File`	prolog source file pathname
`Ln`	logical name

- `allocatedRootObject(Design).`

`Design`	name of the logical Application to distribute

- `allocatedObject(Node,Module).`

`Node`	name of a Virtual Node
`Module`	name of a Module in logical view

Next specific predicates may be generated to take into accounts user's options for each particular action (code extraction, rules checking, document generation):

Predicates for design rules checking:

This predicate specifies which categories of rules have been selected by the user.

- `check(Category,Rules,Result).`

`Category`	name of a rules checker category
`Rules`	prolog rules base file pathname for this category
`Result`	result file pathname for this category

Predicates for code extraction:

These two predicates indicates which source code files have to be generated, and various code generation options (`pragmas`).

- `extract(Component,Kind,Module,Ln,File).`

`Component`	name of a Component or NIL
`Kind`	OPERATION or NIL
`Module`	name of a Module for which code is generated
`Ln`	section logical name suffix (lang::extract_Ln)
`File`	target language source file pathname

- `pragma_xxx(Module,Param_1,..,Param_n).`

`Module`	name of a Module
`Param_i`	value of a pragma parameter

Predicates for documentation generation:

These three predicates specify the list of **ODS** sections to be inserted into documentation, and various user customizable generation parameters.

- `pragma_doc_conf(Parameter,Value).`

`Parameter`	name of a documentation parameter
`Value`	value of specified documentation parameter

- `selectedObject(Module).`

`Module`	name of a Module for which doc must be produced

- `docSection(T,Ln,Pln,L,D,Mod,Title,Contents).`

`T`	Text or File
`Ln`	logical name of section to be inserted into the doc
`Pln`	logical name of higher level section
`L`	level of current section
`D`	TXT, CODE, TXTEND or POSTSCRIPT
`Mod`	name of a selected Module
`Title`	title for current section
`Contents`	text string (T=Text) or file pathname (T=File)

- `graphicBox(Label,X0,Y0,X1,Y1).`

`Label`	Name of a Module
`X0`	top left corner abscissa
`Y0`	top left corner ordinate
`X1`	bottom right corner abscissa
`Y1`	bottom right corner ordinate

- `graphicImp(Pm,Pc,Cm,Cc,View,[Xi],[Yi]).`

`Pm`	Parent Module name
`Pc`	Parent Component name
`Cm`	Child Module name
`Cc`	Child Component name
`View`	OPERATION, TYPE, CONSTANT,...
`[Xi]`	list of segments abscissa
`[Yi]`	list of segments ordinate

- `graphicUse(Cm,Sm,View,Style,[Xi],[Yi],[Lj]).`

`Cm`	Client Module
`Sm`	Server Module
`View`	OPERATION or TYPE
`Style`	1: Uses; 2: Inherits; 3: Attributes
`[Xi]`	list of segments abscissa
`[Yi]`	list of segments ordinate
`[Lj]`	list of flows label

- `graphicState(Module,Label,X0,Y0,X1,Y1).`

`Module`	name of a Module with a STD
`Label`	name of a State
`X0`	top left corner abscissa
`Y0`	top left corner ordinate
`X1`	bottom right corner abscissa
`Y1`	bottom right corner ordinate

- `graphicTrans(Module,Label,Si,Sd,[Xi],[Yi]).`

`Module`	name of a Module with a STD
`Label`	name of a Transition
`Si`	origin State name
`Sd`	destination State name
`[Xi]`	list of segments abscissa
`[Yi]`	list of segments ordinate

1.1.5. Unix interface

STOOD delivery for **Windows** also contains `bash` directory, containing standard **Unix** commands for **PC**. These files come from **Cygnus** company (`http://www.cygnus.com/`), and are not required if another proper version of `cygwin32` or any other implementation of required **Unix** commands have already been installed on your platform.

Executable files contained within `bash` directory should be made accessible in user's execution path. This can be performed by a proper action in local **Windows** environment, or by extending current execution path within `stood.ini` initialization file (refer to § 1.2.6):

```
PATH=%PATH%;$TOOL\bash
```

If this path is not properly set, following alert will be displayed when loading an **Application**:

STOOD uses only a very limited number of **Unix** commands. Next table provides the minimum contents of `bash` directory (or other similar utility) to comply with standard configuration of **STOOD** shell scripts:

`basename.exe`	`hostname.exe`
`bash.exe`	`ln.exe`
`cat.exe`	`ls.exe`
`chmod.exe`	`mkdir.exe`
`cp.exe`	`mv.exe`
`cygwin.dll`	`pwd.exe`
`date.exe`	`rm.exe`
`diff.exe`	`rmdir.exe`
`dirname.exe`	`uname.exe`
`echo.exe`	

More recent versions of `cygwin32` may be available. Tool administrator may update it directly from **Cygnus**, if required. In this case, for compatibility reasons, it may be necessary to recompile `sbprolog` executable file with the new version of `gcc` compiler, or with another compiler. Please note that software contained inside `bash` directory is covered by the **GNU** General Public License (**GPL**)

1.2. User's customizations

STOOD offers numerous capabilities in terms of customizations. Here are described only easy-to-change options or parameters at user's level. They are localized in `.stoodrc` (**Unix**) or `stood.ini` (**Windows**) file. Both files retain the same information, but use a different syntax.

1.2.1. Properties

All these options and parameters may be handled in a generic way by properties organized in categories. To assign a value to a property in a category, operate as follow:

- In `.stoodrc` file (**Unix**):

```
Category.Property1:value1
Category.Property2:value2
```

- In `stood.ini` file (**Windows**):

```
[Category]
Property1=Value1
Property2=Value2
```

These properties may also be set dynamically within the command line.
In this case, you may type:

```
stood Property1=value1 Property2=value2
```

When same properties are set at various locations, they will be taken into account with following priority rules:

- highest priority: command line
- user level: `stood.ini` or `.stoodrc` within working directory
- intermediate: `stood.ini` within `Windows` or `Winnt` directory
- default: `stood.ini` or `.stoodrc` within `bin.xxx` directory

A few internal variables are automatically set by **STOOD** at launch time, and may be used when assigning a value to properties. Note that these internal variables may only be read, and should not be written. These variables are:

`$TOOL`	parent directory of current bin.xxx directory
`$TOOLBIN`	current bin.xxx directory
`$WORKDIR`	current working directory

`$TOOL` and `$TOOLBIN` values take into account the location of the executable file that is actually launched. On Unix especially, it may depend on the way `PATH` environment variable is set.

`$WORKDIR` is the location from where **STOOD** is launched. It is important for the user to have proper file access rights at this level (`rwx`). When launching **STOOD** from a **Windows** shortcut, this location may be specified from appropriate field within shortcut properties dialog box.

1.2.2. Changing Applications search path

STOOD Applications may be stored into several different directories and may be visible from several simultaneous sessions. The way **STOOD** knows where to find them is by reading the contents of `SavePath` property in `Files` category within `stood.ini` or `.stoodrc` file. This variable should contain a list of valid pathnames for current file system, with a few syntactic constraints.

It should be noted that, even under **Windows**, **STOOD** uses **Unix** shell scripts to perform file handling operations. It is thus prohibited to store **Applications** inside directories which name contains invalid characters as regards standard **Unix** files naming rules. Directory names like `Program Files` should be avoided.

A list of directories containing **STOOD Applications** may be defined by assigning a value to `SavePath` property. First path of the list will be used as a default directory when creating new **Projects**. It is a good idea to put a working directory at first position in path list. It is thus likely that proper read and write access will be available when creating new **Projects** and **Applications**. Other pathes in the list may be shared directories on any local or remote file server.

Example:

In `stood.ini` file, typical `SavePath` setting would be:

```
[Files]
SavePath=$WORKDIR,$TOOL\examples,C:\hood\prj1,
\\unix-server\hood\lib
```

In `.stoodrc` file, similar setting would be:

```
Files.SavePath:$WORKDIR,$TOOL/examples,
/users/hood/prj2,/home/unix-server/hood/lib
```

In both cases:

- First path specifies current working directory as default saving area.
- Second path refers to an **Application** examples directory.
- Third path gives access to a local saving directory.
- Fourth path gives access to a remote **Unix** server.

1.2.3. Changing default target languages

STOOD is a multi-languages environment. Several implementation langages may be used at the same time for a same **Project**. That's why standard configuration provide access to **Ada**, **C** and **C++** features at the same time for any **Application**. A **pseudo-code** is also available to perform some specific operations.

Anyway, a "main" language must always be specified, which will be used by default when needed. Standard default language is **Ada**.

It is possible to change these settings by editing `DefaultLanguage` property in `stood.ini` or `.stoodrc` file. On the same way, it is possible to hide information related to some unused languages, by setting `DiscardedLanguages` property. This last feature is mainly helpful to minimize the number of sections appearing within textual editors. These two properties belong to the `General` category.

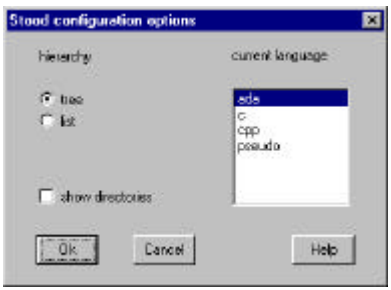
In `stood.ini` file, a possible setting could be:

```
[General]
DefaultLanguage=ada
DiscardedLanguages=c,cpp
```

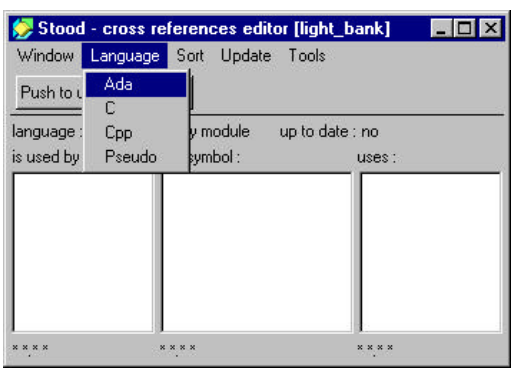
In `.stoodrc` file, same setting would be:

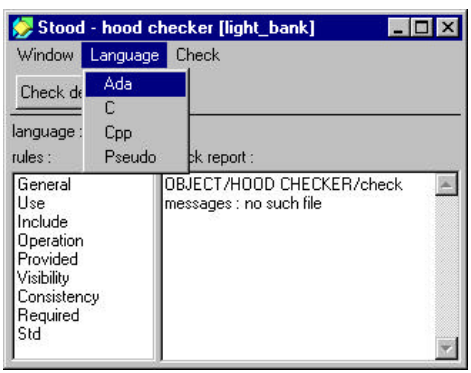
```
General.DefaultLanguage:ada
General.DiscardedLanguages:c,cpp
```

Note that default language may also be changed during an active session by using `options` item of `window` menu of `main editor` (refer to chapter 3.4.4), and that current default language is kept within **Application** storage files:



It is also possible to change default language locally when performing language dependent actions (typically: updating *cross-references table* or *checking design rules*). These local changes are loosed as soon as regarding window is closed.





1.2.4. Customizing windows buttons, tabs and size

Each **STOOD** window is always composed of at least a menu bar, a button bar and a contextual menus appearing when pressing center or right mouse buttons. A few windows also provide a set of tabs. Menu bar and contextual menu are statically defined. On the contrary, button bar and tabs may be fully customized by editing `stood.ini` or `.stoodrc` file. In addition, default opening location and size of each window on the screen may be predefined:

Buttons, tabs and default size need to be defined individually for each window. A dedicated configuration category is attached to each window. The list of available categories is:

`main`	main editor
`syc`	system editor
`gra`	graphical editor
`vna`	allocation editor
`hie`	inheritance tree
`std`	states-transitions diagram editor
`txt`	text editor
`crf`	cross-references table
`chk`	any rule checker
`utr`	call tree
`ext`	code extractor
`rev`	code reversor
`doc`	document editor
`sch`	documentation scheme editor

1.2.4.1. Customizing buttons

To set a value to `Buttons` property, following syntax should be used:

```
Label_1,Menu_1,Item_1;...;Label_n,Menu_n,Item_n
```

Where:

`Label_i`	label to be displayed in the button bar
`Menu_i`	regarding menu number in menu bar
`Item_i`	regarding menu item number in menu bar

Notes:
- To manage long lists of buttons, local variables may be created.
- A specific setting is available to define exclusive buttons.
- Additional semi-colons may be used to increase separation space.
- When label begins with a `*`, then relevant icon (refer to § 1.1.2.6) will be displayed instead of label name.
- Within `.stoodrc` file, windows identifiers (i.e. `main`) should be written in lowercase characters.

Example: to get following button bar in *main editor*,

In `stood.ini` file, regarding setting should be:

```
[Main]
Buttons=Graph.,2,2;Ods.,2,6;;Inher.,2,4
```

In `.stoodrc` file, same setting would be:

```
main.Buttons:Graph.,2,2;Ods.,2,6;;Inher.,2,4
```

- First button calls item 2 (*Graphic editor*) of menu 2 (*Editors*)
- Second button calls item 6 (*Ods text editor*) of menu 2 (*Editors*)
- Third button calls item 4 (*Inheritance tree*) of menu 2 (*Editors*)

1.2.4.2. Customizing tabs

Tabs may be declared in the same way for appropriate windows (*Graphic editor* and *States-Transitions Diagram editor*). In this case, values for `Tabs` property should comply with following syntax:

```
Logical_name_1,...,Logical_name_n
```

Where `Logical_name_i` is an identifier of a section of the **ODS**, as defined within `config/DataBase` configuration file.

Notes:
- Reserved `#main` logical name gives access to general information.
- To manage long lists of tabs, local variables may be created.

Example:

```
TabOb=Pragma,InstPars,DOC11,DOC132,DOC21
TabTy=TypeTxt,ada::TypeDecl,ada::TypeDef
TabCo=CstTxt,ada::CstDecl,ada::CstDef,
TabEx=ExcTxt
TabDa=DataTxt,ada::DataDef
TabOp=OpTxt,pseudo::OpDef,ada::OpDef,
Tabs=#main,$TabOb,$TabTy,$TabCo,$TabOp,$TabEx,$TabDa
```

- **STOOD** automatically hides unappropriate tabs for current selection, following relevant *DataBase* descriptor information (refer to § 1.1.2.7).

Example: to get following tabs in *graphic editor* when an **Operation** is selected:

In `stood.ini` file, regarding setting should be:

```
[Gra]
Tabs=#main,OpTxt,pseudo::OpDef,ada::OpDef
```

In `.stoodrc` file, same setting would be:

```
gra.tabs:#main,OpTxt,pseudo::OpDef,ada::OpDef
```

- First tab edits general information regarding current selection
- Second tab refers to *textual description* section of the **ODS**
- Third tab refers to *pseudo code* section of the **ODS**
- Fourth tab refers to *ada code* section of the **ODS**

1.2.4.3. Customizing default position and size

Default position and size may be declared in the same way for each window. In this case, values for `Position` and `Extent` properties should comply with following syntax:

```
X_axis_coordinate,Y_axis_coordinate
```

Where `coordinates` are specified in pixel. Point (0,0) is located at the top left corner of the screen.

Notes:
- `Position` property specifies top left corner location of the window.
- `Extent` property specifies bottom right corner location of the window.
- negative values are allowed.

Example:

In `stood.ini` file, a possible setting could be:

```
[Main]
Position=0,0
Extent=400,250
```

In `.stoodrc` file, same setting would be:

```
main.Position:0,0
main.Extent:400,250
```

1.2.5. Changing default fonts and colors

In addition to customization capabilities directly provided by the window manager (**Windows** or **Motif**) which are not described here, it is possible to configure some applicative fonts and colors. This configuration will be performed by setting a few properties inside `stood.ini` or `.stoodrc` files. These properties belong to `Fonts` and `Colors` categories respectively.

1.2.5.1. Customizing fonts

Property name for fonts customization are:

`DefaultFont`	font used by default.
`DiagramFont`	font used in graphical diagrams.
`TreeFont`	font used in graphical trees.
`TEXT`	font used in textual ODS sections.
`CODE`	font used in coding ODS sections.

Value for font properties should be a valid font name and size available for current platform. All other fonts (menus, lists, ...) are controled by the window manager, and should be customized by appropriate procedures in **Windows** control panel or **Motif** ressource files.

On **Unix** workstations, a `Stood` **Motif** ressources file for **STOOD** may be created in any of these locations (none is provided with standard delivery):

- `/usr/lib/X11/app-defaults/Stood`
- `$APPLRESDIR/Stood`
- `bin.xxx/Stood`

Example:

A possible `stood.ini` font configuration could be:

```
[Fonts]
DefaultFont=Arial 10
DiagramFont=Comic Sans MS 10
TreeFont=Comic Sans MS 10
TEXT=Times New Roman 14
CODE=Courier New 14
```

A possible `.stoodrc` font configuration could be:

```
Fonts.DefaultFont:helvetica 12
Fonts.DiagramFont:times 12
Fonts.TreeFont:helvetica 12
Fonts.TEXT:times 14
Fonts.CODE:courier 14
```

1.2.5.2. Customizing colors

Property names for color customization are:

`Module`	Module box in HOOD diagrams
`ModuleExport`	exported Module box in HOOD diagrams
`Component`	Component name in HOOD diagrams
`ConnectionUse`	Use relationship in HOOD diagrams
`ConnectionImpl`	Implemented_By link in HOOD diagrams
`ConnectionLabel`	DataFlow, Exception Flows labels
`State`	State box in STD
`Transition`	Transition link in STD
`TransitionLabel`	labels on Transitions in STD

Value for a color property must be a valid **RGB** code. Most commonly used codes are:

`black`	0	0	0
`white`	255	255	255
`grey`	128	128	128
`dark grey`	192	192	192
`red`	255	0	0
`green`	0	255	0
`blue`	0	0	255

All other combinations are of course possible.

Example:

A possible `stood.ini` color configuration could be:

```
[Colors]
Module=0 0 0
ModuleExport=192 192 192
Component=0 192 0
ConnectionUse=0 0 128
ConnectionImpl=192 192 192
ConnectionLabel=0 0 128
State=0 0 128
Transition=255 0 0
TransitionLabel=0 0 255
```

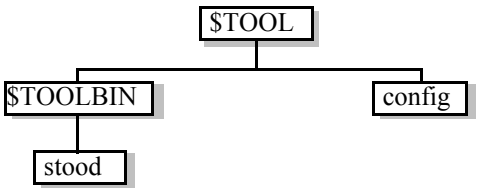
Similar `.stoodrc` color configuration would be:

```
Colors.Module:0 0 0
Colors.ModuleExport:192 192 192
Colors.Component:0 192 0
Colors.ConnectionUse:0 0 128
Colors.ConnectionImpl:192 192 192
Colors.ConnectionLabel:0 0 128
Colors.State:0 0 128
Colors.Transition:255 0 0
Colors.TransitionLabel:0 0 255
```

1.2.6. Customizing environment

A few properties may be changed to customize standard configuration and execution environment of **STOOD**. Changing these properties requires a good knowledge of the way **STOOD** works. It is generally the responsability of a system administrator to customize these properties if needed.

Value of the property `ConfigPath` in `Files` category can be modified to let **STOOD** point to another configuration directory. Default value is `$TOOL/config`, that is `config` directory located in the same parent directory as current `bin.xxx` directory.



When using its internal or external tools, **STOOD** needs to launch **Unix** shell commands (even under **Windows** environment). The `Shell` property in `Shell` category must be set to specify which shell is to be called. Default values are `sh` for **Unix** and `bash` for **Windows**. An additional property specifies whether the shell command window must be displayed or not. Default is `Yes` for this `HideWindow` property.

It is possible to ask **STOOD** to automatically insert an **RCS** tag inside all produced files (except those that are edited by hand) for configuration management purposes. Generic tag value may be specified in `Header` property of `Versioning` category. Default values are blank to specify not to insert a tag, or `$Header$` else.

A few **Unix** environment variables are required by **STOOD** post-processors (rules checkers, code extractors, documentation generators). These variables may be directly set within `Environment` category. Defaults values are `$TOOL/sbprolog` for `STOODPRO` variable, `$TOOLBIN` for `STOODBIN` variable and `$WORKDIR` for `STOODHOME` variable. Note that other **Unix** or **Windows** environment variables may bet set if required. For instance, it may be required to extend locally execution path to give access to specific executable files:

```
PATH=%PATH%;$TOOL\bash
```

Licensing information is also specified by several properties belonging to `Licensing` and `License` categories. Please refer to *Installation Manual* or contact your system administrator or **TNI**'s technical support if you need to set or change these properties:

```
[License]
NFLFile=\\hostname\tools\license\stood.nfl
```

or (exclusive):

```
[Licensing]
Organization=Evaluation
Licensee=None
LicenseCount=1
Mode=Full
ExpirationDate=31/07/2000
Password=6061768
```

1.2.7. Other simple customizations

A set of other properties may be used to customized various additional features of **STOOD**.

- `Welcome` property in `General` category: specifies the string to be displayed on top of *main editor*. Default value is "`Stood 4.1 for HOOD4 & HRT-HOOD`". It is an easy way to identify a particular configuration.

- `ModuleNamePolicy` property in `General` category: specifies the way long names should be truncated within design tree of *main editor*. Possible values are: `CutLeft`, `CutMiddle` and `CutRight`.

- `ModuleNameLength` property in `General` category: specifies the maximum length of names displayed within design tree of *main editor*. If the actual name is longer, it will be truncated in a way specified by `ModuleNamePolicy`.

- `ShowDirectories` property in `General` category: specifies whether **Project** and **Application** names should be displayed by default with their full storage pathname or not. Values are `Yes` or `No`. This property may be changed locally during the session.

- `ModuleHierarchy` property in `main` category: specifies whether design tree of *main editor* should be displayed as a textual list, or a graphical tree. Values are `List` or `Tree`.

- `NewBoxExtent` property in `gra` and `std` categories: specifies the size of newly created boxes in **HOOD** diagram editor and **STD** editor. A typical value is `100 100`.

- `Default` property in `doc` category: specifies which documentation format will be set by default when opening a new *documentation editor*. This default value may be changed locally later during the session. Possible values depend on actually installed document generators, typically: `rtf`, `ps_p`, `mif_p`, `html_p`.

1.2.8. Customizing HOOD

It is possible to extend methodological features thanks to the `hood` category of properties in `stood.ini` or `.stoodrc` files

Currently, only a capability to extend **HOOD Modules** kind is provided. This feature has been applied to add **HRT-HOOD** patterns. To define a new **Module** type, a triplet must be added to `ModuleTypesExt` property. Each triplet has the following form:

```
text_label,db_tag,gra_label
```

- `text_label` is used for relevant **ODS** field and prolog predicates
- `db_tag` may be used in `DataBase` file records `ModuleKind`
- `gra_label` is displayed on top left corner of **HOOD** boxes

Example : extending HOOD4 with HRT-HOOD Module kinds:

In `stood.ini` file:

```
[Hood]
ModuleTypesExt=PROTECTED,pr,Pr;CYCLIC,cy,Cy;
    SPORADIC,sp,Sp
```

In `.stoodrc` file:

```
Hood.ModuleTypesExt:PROTECTED,pr,Pr;CYCLIC,cy,Cy;
    SPORADIC,sp,Sp
```

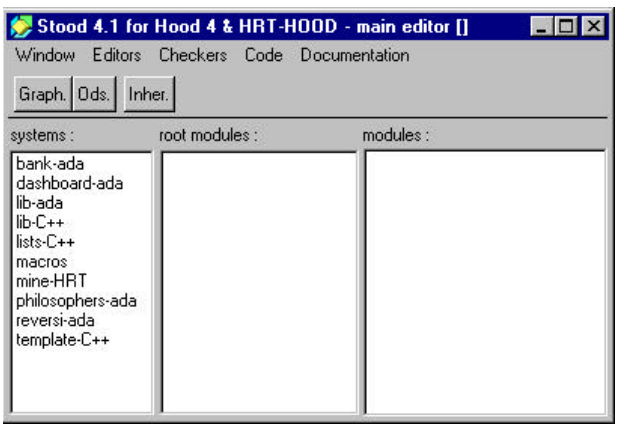
1.3. Launching STOOD

STOOD may be launched in four different modes:

- interactive mode (usual mode)
- semi-interactive mode
- batch mode
- remote control mode (**Unix** only)

Interactive mode is the one which requires standard use of terminal keyboard and mouse. Three last modes require the ability to write command lines instead of using window manager control interface interactively. These command lines should be written with a specific syntax, in a language called **STShell**, and is used to define an Application Programming Interface (**API**) for **STOOD**.

When **STOOD** is launched, an instance of *main editor* is displayed on the screen. Please refer to §3 to get detailed informations about *main editor* contents and use.:



1.3.1. STShell

STShell expressions consist in the invocation of a command, with a list of parameters. General syntax is:

```
Command("parameter1", "parameter2",...)
```

STShell expressions may be either inserted sequentially in macro-commands files (files with a suffix `.sts`), either be sent directly to an active session of **STOOD**, in remote control mode.

1.3.1.1. STShell parameters

Parameters are always strings delimited by double quote characters. This separator may be omitted in following cases:

- for simple identifiers: `{[a..z]|[A..Z]|[0..9]}`
- for integers

Use of `*` wildcard character is allowed. It replaces any sequence of characters. Take care to avoid its use when there is a risk of ambiguity.

Parameters may need to reference a specific window of **STOOD** (browsers, graphical editors, dialog boxes,...). In this case they must match relevant window predefined identifier. Following table provides the list of recognized identifiers.

Note that these identifiers are the same as those used to define help files and to customize buttons and tabs in `stood.ini` or `.stoodrc` files.

`main`	main editor
`syc`	system editor
`gra`	graphical editor
`vna`	allocation editor
`hie`	inheritance tree
`std`	states-transitions diagram editor
`txt`	text editor
`chk`	any rule checker
`crf`	cross-references table
`utr`	call tree
`ext`	code extractor
`rev`	code reversor
`doc`	document editor
`sch`	documentation scheme editor
`dbcfg`	options dialog box
`dbobj`	module selection dialog box
`dbobjla`	module and language selection dialog box
`dbcompare`	designs comparison dialog box
`dbcopy`	design copy dialog box
`dbreplace`	design replace dialog box
`last`	last opened window

Parameters may also need to reference a list in a browser. Each list is identified by an integer index, as follow:

• *main editor*:

`1`	systems :
`2`	root modules :
`3`	modules :

• *text editors*:

1	modules :
2	sections :
3	components list
4	symbol table

• *cross-references tables*:

1	this symbol : :
2	is used by :
3	uses :

• *code extractors*:

1	modules :
2	pragmas :
3	pragma parameters :

• *document editors*:

1	configuration :
2	modules :
3	schemes :

Notes:

- Only one window of each kind may be referenced at a time within a sequence of **STShell** commands.
- All parameters referencing menu, menu item, list, list element and button names should match *exactly* names, displayed within **STOOD** windows.

1.3.1.2. STShell commands

Following commands are available to build **STShell** programs. These commands generally represent a basic interaction with windows components (lists, menus, buttons,...). A few commands represent a higher level shortcut to perform some predefined actions.

- `Exec("filename")` : execute **STShell** program contained in file given as parameter. This file should contain a list of valid **STShell** expressions.

- `Context("project","application")` : select given **Project** and **Application** within *main editor*. This command is a shortcut of following sequence:

```
ListSelect(main,1,"project")
ListSelect(main,2,"application")
```

- `Menu("id","menu","item")` : execute a given item of a window menubar.

`id`	window identifier
`menu`	Menu name in window menu bar
`item`	Item name in menu

- `Button("id","label")` : "press" specified button of a window button bar. May be used as a shortcut for the `Menu` command. For instance, `Button(main,"graph.")` as the same effect as `Menu(main,editors,"graphic editor")`, if `graph.` button was properly defined in initialization file.

`id`	window identifier
`label`	Button label in button bar

- `ListSelect("id","list","element")` : select given element in a list of a window.

`id`	window identifier
`list`	list index (1, if there is only one list)
`element`	Element name in the list

- `ListMenu("id","list","item")` : execute a given item of a contextual menu in a list of a window.

`id`	window identifier
`list`	list index (1, if there is only one list)
`item`	item name in contextual menu

- `Answer("value")` : fill in an active dialog box with given string.

- `Click("id","label")` : “press” a built-in button of a window. This command should not be used for customizable buttons within window button bar. In this case, use `Button` command.

`id`	window identifier
`label`	button label

- `Ok`, `Cancel`, `Yes`, `No` : “press” corresponding button in a simple dialog box. May be used as shortcuts for `Click(last,ok)`, ...

- `System("OS command")` : executes specified external command, which is supposed to be recognized by current executing environment.

1.3.1.3. STShell program example

```
#------------------------------
# CODE EXTRACTION MACRO EXAMPLE
# stood v4.1 - TNI - August 1999
#------------------------------
#
# Select "test" design inside "tests" system :
Context(tests,test)
#
# Open "ada extractor" from "main editor" :
Menu(main,code,"ada extractor")
#
# Launch ada code extraction :
Menu(ext,extract,"full extraction")
Click(dbobj,OK)
#
# Quit "ada extractor" :
Menu(ext,window,quit)
#
# Open "ada text editor" from "main editor" :
Menu(main,editors,"ada text editor")
#
# Show "extraction messages" :
ListSelect(txt,1,test)
ListSelect(txt,2,"extraction messages")
#
# Don't exit to let last window open.
```

Other macro-commands examples may be found in `exMacros` directory, provided with standard delivery.

1.3.2. STOOD executing modes

In order to be able to launch **STOOD**, first check that used **Windows** shortcut or **Unix** execution path is set properly. They should point to **STOOD** binary files directory (refer to §1.1.1)

1.3.2.1. Interactive mode

When launching **STOOD** without any option, an interactive session is started. The tool may thus be controled with the keyboard and the mouse of user's terminal. In interactive mode, a license token is used for each active session. To launch **STOOD** in interactive mode, just double-click on relevant **Windows** shortcut or, on your **Unix** terminal, enter:

```
stood
```

1.3.2.2. Semi-interactive mode

This mode is useful to preset **STOOD** in a predefined configuration, and then let the user go on working in interactive mode. Predefined configuration should be described by a sequence of **STShell** expressions in a `.sts` file. The user may thus launch:

```
stood -f filename.sts
```

1.3.2.3. Batch mode

The aim of this executing mode is to let **STOOD** perform actions without any user direct interaction. It is typically the way to launch code and documentation generation for a stored **Application**. This mode also requires a **STShell** command file, to describe operations to be performed, but unlike semi-interactive mode, no license token is required, and **STOOD** will close automatically at the end of commands sequence. To launch **STOOD** in batch mode:

```
stood -batch -f filename.sts
```

Note that for implementation reasons, on **Unix** platforms, `DISPLAY` environment variable should be set, even in batch mode.

1.3.2.4. Remote control mode

On **Unix** platforms only, it is possible to send **STShell** commands to an active session of **STOOD**. An input pipe is automatically created when **STOOD** is launched. This pipe is always named `st` and is located in current working directory.

STShell expressions may then be sent to this file with usual Unix commands:

```
echo 'Context(project,application)' > st
echo 'Menu(main,editors,"graphic*")' > st
echo 'ListSelect(crf,1,"*oper1")' > st
cat macros.sts > st
...
```

Notes:

- It is not possible to send commands to a remotely mounted `st` file. If your working directory is remote, you must `rlogin` on relevant file server, to be able to get access to the pipe.
- Take care to get write rights on your current working directory, else **STOOD** will not be able to create `st` file.

Associated to the capability to customize *external tools*, remote control mode is the prefered way to let **STOOD** interact with other tools in a software development environment.

2. Projects and Applications

This section provides informations about the concepts of **Projects** and **Applications** in **STOOD**. Two input formats that can be used to import a full new **Application**, or to update an existing one, are also described,

2.1. Projects

While working with **STOOD**, your current **Project** is generally represented by a set of cooperative **Applications** (refer to §2.2). Small autonomous **Projects** may contain only one **Application**. On the contrary, large **Projects** or/and **Projects** reusing previously defined libraries will be more easily managed if composed of separate **Applications**.

Projects are sometimes also called **Systems**, as they directly refer to **HOOD System Configuration** concept. Main benefits of such a high level organization for the **Project** are:

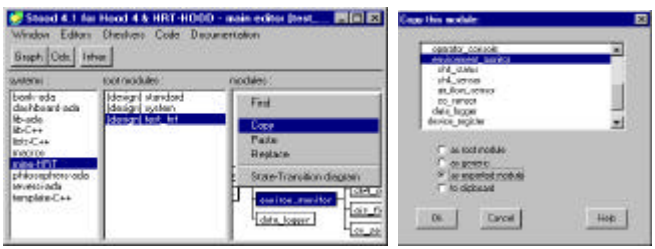
- **Project** organization may map team organization.
- **Project** organization may map pre-defined implementation constraints.
- Reusable **Applications** may be easily shared between several **Projects**.
- Parts of the **Project**, not developed with **HOOD**, may appear there anyway, to highlight common interfaces with **HOOD Applications**.

A mix of these options is of course possible, and it is the responsability of the project manager to decide which organization is optimal for the system.

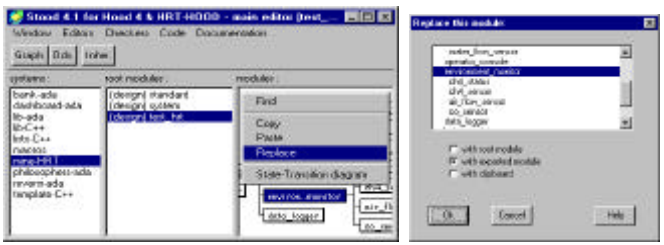
For **STOOD**, following terms are synonyms:

- **Project**
- **System**
- **System Configuration**

From current **Application** point of view, all other **Applications** belonging to the same **Project**, are called **Environment Modules**. **System Configuration** must comply with operational constraints during the whole development phase. In order to be able to follow organizational changes, **STOOD** provides *import* and *export* functions to control **Project** organization during development. To export a branch of current **Design Tree**, use *Copy* in pop-up menu of *modules* area in *main editor*:

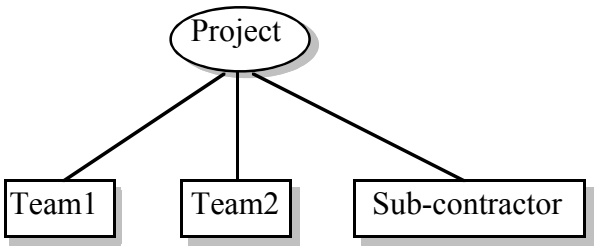


To re-import a previously exported branch of current **Design Tree**, use *Replace* in pop-up menu of *modules* area in *main editor*:



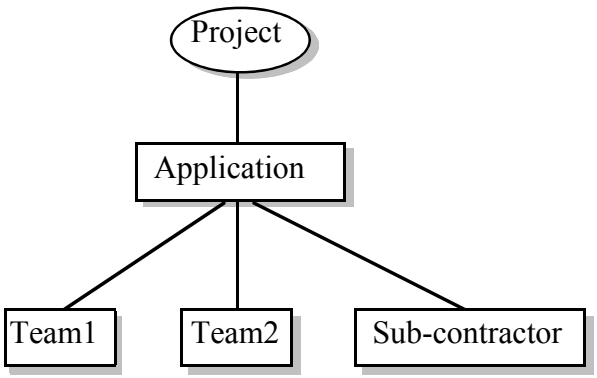
2.1.1. Mapping team organization

In the context of a **Project** needing to be developed by several partners (project manager, designers, programmers, sub-contractors, ...), **System Configuration** may be set up in a such way that it fits team organization. Each member of the team will then be able to work on a restricted part of the project.



Project manager need to specify clearly the interface of each sub-system, which will be formalized by the **Provided Interface** of each **Root Module**.

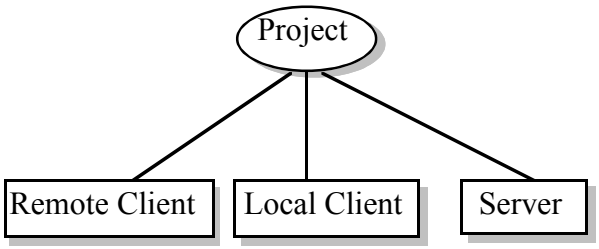
An import of all the sub-systems into a unique consolidated **Application** may be performed for integration phase. **HOOD** visibility rules ensure that this operation is feasible at any time, if interfaces have been left unchanged.



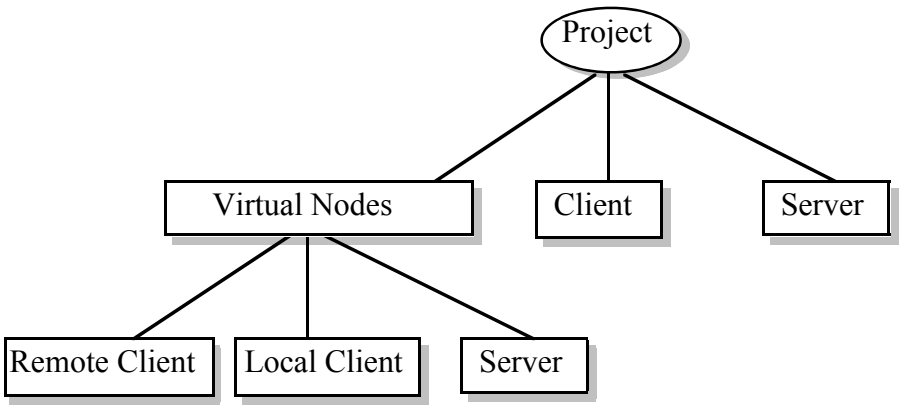
In a similar way, export function may be used to dispatch a development into several sub-systems, without breaking **Application** overall consistency.

2.1.2. Mapping pre-defined implementation constraints

Sometimes, project architecture needs to be driven by implementation constraints (available hardware, cards, processors, ...), in which case it may be required to take into account these input constraints to set up current **System Configuration**. By this way, software functionnalities can be pre-allocated to physical parts of the system to be developed. This situation often occurs for distributed and/or embedded **Applications** developments.



In this example, three logical **Applications** need to be developed to match the three different kinds of hardware that will be deployed. Developing software with **HOOD**, it is also possible to postpone allocation of logical **Applications** onto hardware elements, thanks to **Virtual Nodes**:



In this case, only two logical **Applications** need to be developed. In addition, a *virtual* definition of hardware architecture is provided in a **Virtual Node** description, attached to the **Project**. Several **Virtual Nodes** descriptions may be defined for a same logical development. Final result may be controled by an allocation table up to the end:

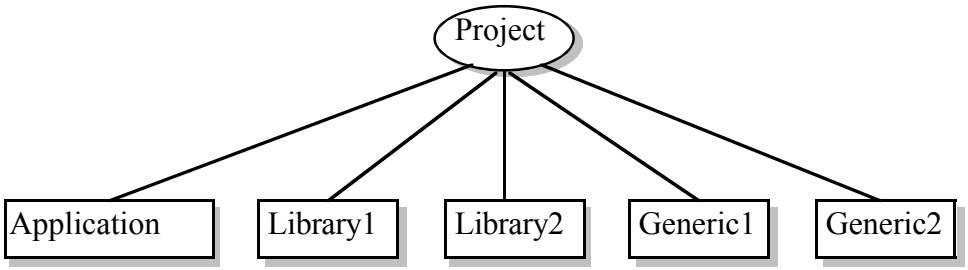
VN allocation table	Client	Server
Server		x
Local Client	x	
Remote Client	x	

In **STOOD**, the *allocation editor* may be used to perform **Application** deployment to a distributed system (refer to § 5).

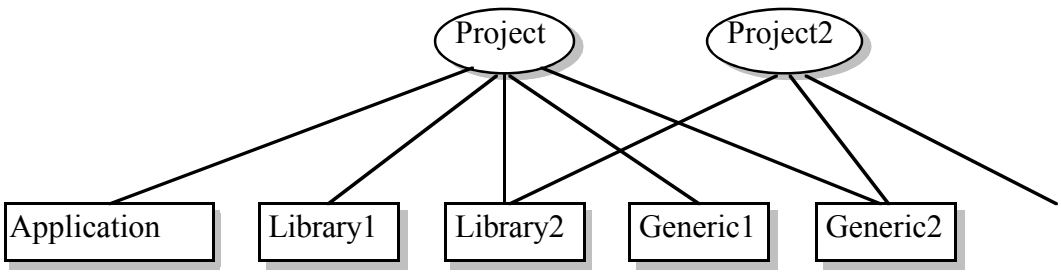
2.1.3. Referencing reusable applications

Another use of the **System Configuration** is to specify which existing **Applications**, software components or components libraries are to be used within current development context. In this context, specified sub-systems won't need to be modified, we only need a description of *what* services they provide, and not *how* they are implemented. This partial view of an **Application** is called the **Provided Interface** of relevant **Root Module**.

With **HOOD**, reusable components may also be **Generic Applications**. That means that some parameters need to be fully defined before the **Generic** is actually used. A specific kind of **HOOD Modules**, called **Instance Of**, should be used for this purpose.



Libraries and **Generics** may be shared between several **Projects**:

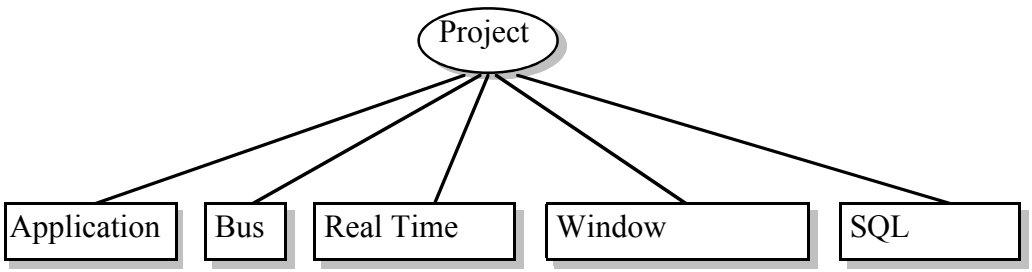


It is also possible to create a new library or **Generic** library by exporting a **Module** already defined somewhere inside **Application** hierarchy.

2.1.4. Referencing external sub-systems

When a sub-system is not or cannot be designed with **HOOD**, it may be represented anyway by a *dummy* **Application**. Like libraries, only its **Provided Interface** will be available in the context of current **Project**. Usual cases are:

- Interfacing with existing hardware (memory addresses, interrupt vectors, device drivers, ...).
- Interfacing with **Operating System** facilities (file system, tools, real-time executive, ...).
- Interfacing with a program developped in another way (for instance code generated from an interactive User Interface builder tool).
- Interfacing with standard language libraries (stdio, stdlib, ..., **Ada** standard libraries, ...).



Thanks to this capability to separate interface and implementation, it is possible to use **HOOD** as a prefered technique to perform system integration. It is not required to develop all the sub-systems with **HOOD**, but in any cases, a precise and formal description of each **Provided Interface** will increase overall **Project** development quality level.

2.1.5. Storing a System Configuration

For **STOOD**, a **Project** simply consists in a list of references to **STOOD Applications**. The three possible kinds of **HOOD Applications** are:

- **Root Modules** (called here root objects for **HOOD** 3 compatibility reasons)
- **Generics**
- **Virtual Nodes**

This list is stored in a single file with a `.syc` suffix. Simplest way to control contents of this file is to use a *system editor*.

```
SYSTEM_CONFIGURATION IS

    ROOT_OBJECTS
         --|E:\stood\tutorial\dispatching|--,
         --|E:\stood\tutorial\drawings|--,
         --|..\exAda95\STANDARD|--,
         --|..\exAda95\TEXT_IO|--

    GENERIC
         --|E:\stood\tutorial\list|--,
         --|E:\stood\tutorial\stack|--

    VIRTUAL_NODES
         --|\\server\system\architecture1|--

END
```

Pathnames found within `.syc` files are built with one member of `SavePath` configuration variable as basename, and **Application** name as filename. **STOOD** is able to equally recognize **UNIX** and **Windows** file pathnames. *Main editor* (refer to §3) and *system editor* (refer to §4) should be used to manage **Projects**.

Main actions that can be performed on a **Project** are:

- Create a new **Project**
- Select an existing **Project**
- Delete an existing **Project**
- Modify contents of a **Project**

2.2. Applications

After having selected a **Project**, the user must specify the **Application** on which (s)he intends to work. To comply with **HOOD** concepts, an **Application** in **STOOD** represents a hierarchy of **Modules**. At high level it can be handled by the **Root Module** of this hierarchy.

A given **Application** may be share by several **Projects**, but only one user may modify a given **Root Module** at a time. Other users may open it, but they will be limited to read-only operations. Within **HOOD** literature, an **Application** is also generally called a **Design**. It is also possible to define parametric **Applications**, called **Generics**.

For **STOOD**, following terms are synonyms:

- **Application**
- **Root Module**
- **Design** or **Generic**

2.2.1. Kinds of Applications

HOOD defines three different kinds of **Applications**:

- **Designs**
- **Generics**
- **Virtual Nodes**

Designs are the most common kind of **Applications**. They can be used to produce source code files to be compiled and linked in order to get executable or at least linkable files at the end (that is a runnable program or a library).

Generics are parametric **Applications** to be used at design level. When similar feature is available with implementation language (i.e. generic packages in **Ada** or templates in **C++**), it is an efficient way to create, document and use reusable components.

While **Designs** and **Generics** must be used to describe logical view of the system, **Virtual Nodes** may be used to specify physical implementation constraints. This is particularly useful for distributed systems, for which, not one but several executable files must be created from a given logical design. **HOOD Virtual Nodes** tree represents this particular software organization, and is linked to a **Design** by an allocation table (refer to §2.4).

2.2.2. Applications storage

STOOD doesn't require any proprietary database. Each **Application** is stored in a dedicated directory, and access to retained information may be performed thanks to usual **Operating System** commands.

In order to make this storage compatible between **Unix** and **Dos** systems, a **Unix** utility is used on **Windows** platforms. For **Windows** users, it is thus possible to indifferently access **Applications** stored locally on their **PC**, or shared by any **Unix** server platform, if a proper gateway is available.

2.2.2.1. Standard storage organization for an Application

Under standard configuration of storage area, a **STOOD Application** is fully contained in a unique directory of the same name. This main directory encompasses following elements:

- A set of global files for the **Application**:

`Stood.dg`	architecture and graphical objects
`Stood.typ`	specifies Design, Generic or VN
`Stood.sp`	external view of the architecture
`Stood.pro`	prolog description of the interface
`Stood.doc`	storage of documentation options
`Stood.sih`	header for Standard Interchange Format
`Bak.dg`	previous version of Stood.dg
`Bak.typ`	previous version of Stood.typ
`Bak.sp`	previous version of Stood.sp

• A set of subdirectories containing results of processing actions:

`_trash`	temporary files
`_doc`	produced documentation files
`_doc_schemes`	local documentation schemes
`_checks`	produced verification reports
`_ada`	produded Ada files
`_c`	produced C files
`_cpp`	produced C++ files

• A subdirectory for each **Module** of the **Application**

2.2.2.2. Standard storage organization for a Module

Each **Module** subdirectory contains information related to this part of the **Application** only. Stored files may contain documentation (informal text), **HOOD** or **STOOD** specific information (with related specific syntax), or target language code (**Ada**, **C** or **C++**).

Under standard configuration, subdirectory contents for a **Module** looks like this (it is likely that all these files will never be present at the same time):

- A set of global files for the **Module**:

`PRAGMA`	list of HOOD pragmas
`instpars.st`	symbols for instance parameters
`specHeader.u`	file header for Ada spec file
`specHeader.u_st`	associated symbols table
`specHeader.c`	file header for C spec file
`specHeader.c_st`	associated symbols table
`specHeader.cc`	file header for C++ spec file
`specHeader.cc_st`	associated symbols table
`bodyHeader.u`	file header for Ada body file
`bodyHeader.u_st`	associated symbols table
`bodyHeader.c`	file header for C body file
`bodyHeader.c_st`	associated symbols table
`bodyHeader.cc`	file header for C++ body file
`bodyHeader.cc_st`	associated symbols table
`init.u`	Ada initialization bloc
`init.u_st`	associated symbols table
`modif`	list of modifications

• A subdirectory containing the **Tests** sequences: `OTS`

`test_desc.t`	textual description
`test_sequ.u`	Ada code for Test sequence

• A subdirectory containing the description files: `DOC`

`StaPro.t`	statement of the problem
`RefDoc.t`	referenced documents
`StrReq.t`	structural requirements
`FunReq.t`	functional requirements
`BehReq.t`	behavioural requirements
`ParDes.t`	parent description
`UseMan.t`	user manual outline
`GenStr.t`	general strategy
`IdeChi.t`	identification of children
`IdeStr.t`	identification of data structures
`IdeOpe.t`	identification of operations
`GroOpe.t`	grouping operations
`IdeBeh.t`	identification of local behaviour
`JusDes.t`	justification of design decisions
`ImpCon.t`	implementation constraints
`header`	informal header for code files
`CeiPri.hrt`	HRT-HOOD Ceiling Priority
`Period.hrt`	HRT-HOOD Period
`Offset.hrt`	HRT-HOOD Offset
`MinTim.hrt`	HRT-HOOD Minimum Arrival Time
`MaxFreq.hrt`	HRT-HOOD Maximum Frequency
`Ddline.hrt`	HRT-HOOD Deadline
`Priori.hrt`	HRT-HOOD Priority
`PreCon.hrt`	HRT-HOOD Precedence Constraints
`TimTra.hrt`	HRT-HOOD Time Transformation
`Import.hrt`	HRT-HOOD Importance

• A subdirectory containing all the **Operations**: `OP`

`operation.t`	text. description of declaration
`operation.s_st`	symbols table of declaration
`operation.t2`	description of implementation
`operation.hx`	list of handled exceptions
`operation.x`	Ada code extension
`operation.x_st`	associated symbols table
`operation.p`	pseudo-code
`operation.p_st`	associated symbols table
`operation.u`	Ada code
`operation.u_st`	associated symbols table
`operation.c`	C code
`operation.c_st`	associated symbols table
`operation.cc`	C++ code
`operation.cc_st`	associated symbols table
`operation_test.t`	description of test
`operation_prec.t`	description of test preconditions
`operation_prec.u`	Ada code of test preconditions
`operation_post.t`	description of test postconditions
`operation_post.u`	Ada code of test postconditions
`operation_modif`	description of modifications
`operation_header.u`	header for Ada subunit
`operation_header.u_st`	associated symbols table
`operation_budg.hrt`	HRT-HOOD Operation budget
`operation_wcet.hrt`	HRT-HOOD Operation WCET

• A subdirectory containing all the **Types**: `T`

`type.t`	textual description
`type.s`	Ada incomplete declaration
`type.s_st`	associated symbols table
`type.u`	Ada full declaration
`type.u_st`	associated symbols table
`type.h`	C declaration
`type.h_st`	associated symbols table
`type.hh`	C++ declaration
`type.hh_st`	associated symbols table

• A subdirectory containing all the **Constants**: `C`

`constant.t`	textual description
`constant.s`	Ada incomplete declaration
`constant.s_st`	associated symbols table
`constant.u`	Ada full definition
`constant.u_st`	associated symbols table
`constant.h`	C definition
`constant.h_st`	associated symbols table
`constant.hh`	C++ definition
`constant.hh_st`	associated symbols table

• A subdirectory containing all the **Exceptions**: `X`

`exception.t`	textual description

• A subdirectory containing all **Operation Sets**: `OPS`

`op-set_name.t`	textual description

• A subdirectory containing all the **Data**: `D`

`data.t`	textual description
`data.s`	Ada definition
`data.s_st`	associated symbols table
`data.c`	C definition
`data.c_st`	associated symbols table
`data.cc`	C++ definition
`data.cc_st`	associated symbols table

• A subdirectory containing behavioural information: `STD`

OBCS code:

`obcs.t`	textual description of interface
`obcs.t2`	textual description of implementation
`obcs.p`	pseudo-code
`obcs.p_st`	associated symbols table
`obcs.u`	Ada code
`obcs.u_st`	associated symbols table
`obcs.c`	C code
`obcs.c_st`	associated symbols table
`obcs.cc`	C++ code
`obcs.cc_st`	associated symbols table
`OBCS_header.u`	header for Ada separate subunit
`OBCS_header.u_st`	associated symbols table

States:

`state.t`	textual description
`state_set.u`	Ada code for assignment
`state_set.u_st`	associated symbols table
`state_get.u`	Ada code for testing
`state_get.u_st`	associated symbols table
`state_set.c`	C code for assignment
`state_set.c_st`	associated symbols table
`state_get.c`	C code for testing
`state_get.c_st`	associated symbols table
`state_set.cc`	C++ code for assignment
`state_set.cc_st`	associated symbols table
`state_get.cc`	C++ code for testing
`state_get.cc_st`	associated symbols table

Transitions:

`transition.t2`	textual description
`transition_cnd.u`	Ada code for condition
`transition_cnd.u_st`	associated symbols table
`transition_exc.u`	Ada code for refusal
`transition_exc.u_st`	associated symbols table
`transition_cnd.c`	C code for condition
`transition_cnd.c_st`	associated symbols table
`transition_exc.c`	C code for refusal
`transition_exc.c_st`	associated symbols table
`transition_cnd.cc`	C++ code for condition
`transition_cnd.cc_st`	associated symbols table
`transition_exc.cc`	C++ code for refusal
`transition_exc.cc_st`	associated symbols table

2.2.2.3. Changing standard storage configuration

If this standard storage organization doesn't fit company's, project's or users's requirements, it is possible to customize filenames and locations by editing `config/DataBase` configuration file.

This file contains a long list of section descriptors. For sections that are stored in a file, the field describing storage location looks like this:

```
text 'D1/D2/.../Dn/filename'
```

Please note that separator is always a / even for **DOS** platforms.

Each element of the pathname may be a valid constant string or refer to the actual value of a pseudo-variable. Refer to §1.1.2.7 for more detailed information.

Example:

Standard definition of storage file for the textual description of an **Operation** is:

```
$Ho/$Dg/$Ob/OP/$Op.t
```

With following actual values:

```
$Ho=/home/users/hood-designs
$Dg=appli1
$Ob=control
$Op=start
```

Actual pathname becomes:

```
/home/users/hood-designs/appli1/control/start.t
```

2.3. SIF and CPF

In addition to **STShell** language (refer to § 1.3.1), it is possible to create or update an **Application** by loading a file containing either a descriptor written in **HOOD** Standard Interchange Format (**SIF**), either a list of instructions written in Change Propagation Format (**CPF**).

2.3.1. Standard Interchange Format

Standard **I**nterchange **F**ormat is formally described in **HOOD R**eference **M**anual. **STOOD SIF** generator and interpreter complies with **SIF** v4 described in **HRM 4**. This syntax is highly upwards compatible with previous version.

SIF was initially defined to make **Application** exchanges between different **HOOD** tools possible. It is more generally used as an input/output text file format for exchanges between a **HOOD** tool and other software of development environment. For instance, following programs work with a **SIF** interface:

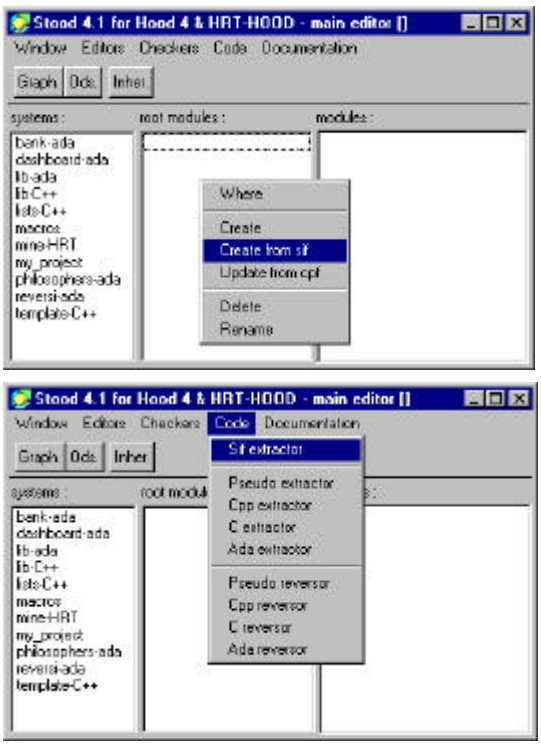
- **HOOD Checker**: checks consistency between a set of **Ada** source files and **SIF** files.
- **Ada2HOOD**: **Ada** reverse engineering utility. It takes **Ada** packages as input and produces a **SIF** file as output.

It may also be used as a storage format for **STOOD Applications**, but this is not recommended, because there may be a loss of information, and other simpler techniques may provide better result.

As **STOOD Applications** are stored in a unique well identified directory, it is easy to create an archive file with standard **Operating System** utilities.

SIF should also be considered as a formal definition of the **HOOD O**bject **D**escription **S**keleton (**ODS**), which is the reference frame for detailed design activity, and for generation of standard documentation.

STOOD provides a command to generate a **SIF** file for the root or a single branch of current **Application** design tree, and another command to create a new **Application** from a given **SIF** file. Please read **HRM** 4 to get detailed information about **SIF** 4 syntax.

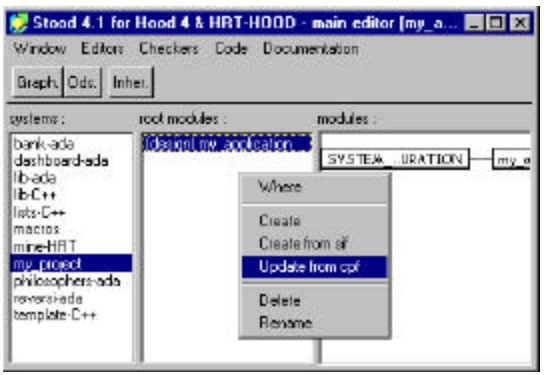


2.3.2. Change Propagation Format

Change **P**ropagation **F**ormat (**CPF**) was defined to meet incremental reverse engineering requirements. The aim is to update an existing **HOOD Application** from identified changes in relevant **Ada** code. Anyway, it may be used for other similar purposes.

With **CPF** instructions, it is only possible to create or delete **Components** in *existing* **Terminal Modules**. There is no way to control the **HOOD** hierarchy with this technique.

To update an existing **Application** from a **CPF** file, use appropriate pop-up menu within *root modules* area in *main editor*:



2.3.2.1. Creating Components

To create a new **Component** in an existing **Terminal Module**, one of the following instructions may be used:

```
New_Type_Or_Subtype module.type_name
Location Specification|Private|Body
[ Discriminant_Part discriminant_part ]
[   Type_Definition type_definition
  | Subtype_Indication subtype_indication  ]

New_Private_Type module.type_name
[ Discriminant_Part discriminant_part ]
Type_Definition type_definition

New_Limited_Private_Type module.type_name
[ Discriminant_Part discriminant_part ]
Type_Definition type_definition

New_Number module.constant_name
Location Specification|Private|Body
Value expression

New_Constant module.constant_name
Location Specification|Private|Body
Subtype_Indication   subtype_indication
          | constrained_array_definition
Value expression
```

```
New_Variable module.data_name
Location Specification|Private|Body
Subtype_Indication   subtype_indication
          | constrained_array_definition
Value expression

New_Exception module.exception_name
Location Specification|Private|Body

New_Procedure module.operation_name
[ Parameter_Types type_mark {,type_mark } ]
Location Specification|Private|Body
{ Name identifier  Mode In|In Out|Out
  [ Value expression ] }
[ declarative_part ]
Begin sequence_of_statements
[ Exception handler { handler } ]
End

New_Function module.operation_name
Result_Type type_mark
[ Parameter_Types type_mark {,type_mark } ]
Location Specification|Private|Body
{ Name identifier  Mode In|In Out|Out
  [ Value expression ] }
[ declarative_part ]
Begin sequence_of_statements
[ Exception handler { handler } ]
End
```

2.3.2.2. Deleting Components

To delete an existing **Component** in a **Terminal Module**, one of the following instructions may be used:

```
Deleted_Type_Or_Subtype module.type_name

Deleted_Constant module.constant_name

Deleted_Number module.constant_name

Deleted_Variable module.data_name

Deleted_Procedure module.operation_name
[ Parameter_Types type_mark {, type_mark } ]

Deleted_Function module.operation_name
Result_Type type_mark
[ Parameter_Types type_mark {, type_mark } ]

Deleted_Exception module.exception_name
```

2.3.2.3. Example of CPF instructions file

- Create a new **Provided** typed **Constant** from following **Ada** code:

```
E : constant FLOAT := 2.7182818;
```

```
New_Constant Terminal.E
Location Specification
Subtype_Indication FLOAT
Value --|2.7182818|--
```

- Create a new **Internal** numeric **Constant** from following **Ada** code:

```
Pi : constant := 3.1415927;
```

```
New_Number Terminal.Pi
Location Body
Value --|3.1415927|--
```

- Create new **Data** from following **Ada** code:

```
Var : FLOAT := 0.0;
```

```
New_Variable Terminal.Var
Location Body
Subtype_Indication FLOAT
Value --|0.0|--
```

• Create a new **Internal Type** from following **Ada** code:

```
type Typ (max : INTEGER) is record
  x : INTEGER;
  y : INTEGER;
end record;
```

```
New_Type_Or_Subtype Terminal.Typ
Location Body
Discriminant_Part --|(max : INTEGER)|--
Type_Definition --|record
    x : INTEGER;
    y : INTEGER;
end record|--
```

• Create a new **Provided Operation** from following **Ada** code:

```
function Op(
  x : in INTEGER := 0;
  y : in CHARACTER) return FLOAT is
begin
  x := x + 1;
  return 0.0;
end Op;
```

```
New_Function Terminal.Op
Result_Type FLOAT
Parameter_Types INTEGER, CHARACTER
Location Specification
Name x Mode In Value 0
Name y Mode In
Begin --|x := x + 1;
  return 0.0|-- End
```

• Create a new **Provided Exception** from following **Ada** code:

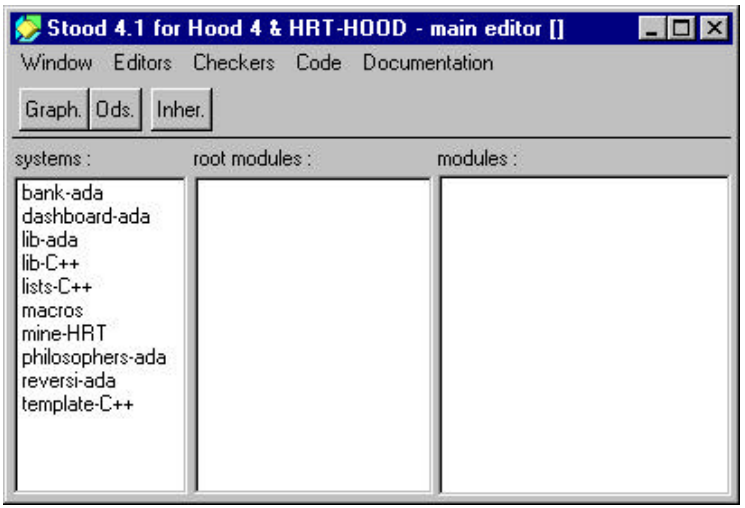
```
Ex : exception;
```

```
New_Exception Terminal.Ex
Location Specification
```

• And now, let's delete all these new **Components**:

```
Deleted_Constant Terminal.E

Deleted_Number Terminal.Pi

Deleted_Variable Terminal.Var

Deleted_Type_Or_Subtype Terminal.Typ

Deleted_Function Terminal.Op
Result_Type FLOAT
Parameter_Types INTEGER, CHARACTER

Deleted_Exception Terminal.Ex
```

3. Main editor

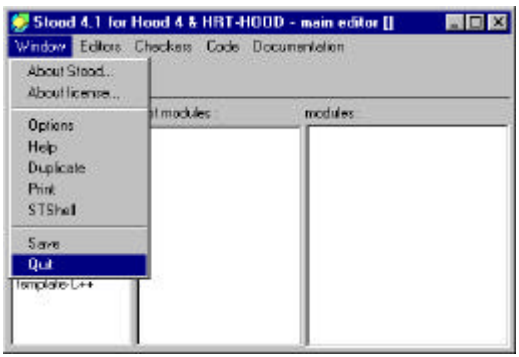


As soon as **STOOD** is launched (refer to §1.3.2), a *main editor* opens out. This window will be kept open until the session is closed. When entering a session, a license token is used; when leaving the session, this token is released.

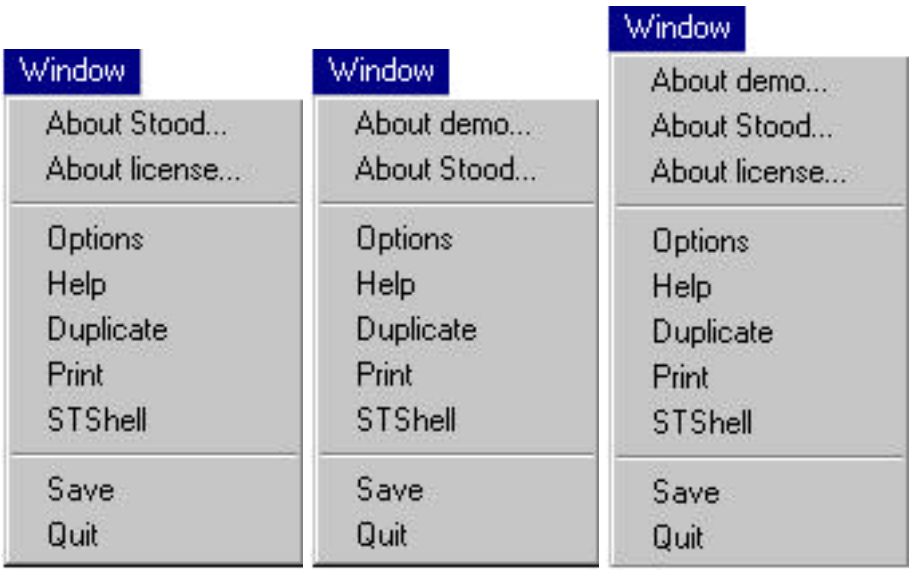
main editor is composed of three selection areas, a button bar and a menu bar. Title of *main editor* may be customized by setting `Welcome` property within initialization file (refer to §1.2.7).

Selection areas are used to select current **Project** (refer to §2.1) and current **Application** (refer to §2.2). Menu bar provides a direct access to editing windows, and to each post-processor

It is also possible to open other main editors without leaving current session. This can be useful to work on several **Root Modules** simultaneously. To close *main editors* (and thus close the session), use *Quit* command of *Window* menu:



3.1. Session management



Menu *Window* of *main editor* should be used to manage **STOOD** sessions. It provides general functions such as licensing information, options setting, saving and quitting. Upper part of this menu may change as regards licensing mode. It may look like one of the three pictures presented above.

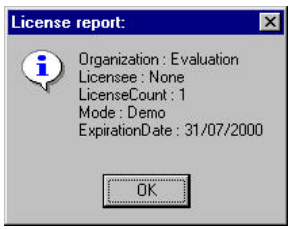
About demo... : This command only appears when STOOD runs on demonstration mode. It recalls demonstration limitations:

About Stood... : This command provides general information about current version of **STOOD**:

About license... : This command should be used to get information about installed and used license tokens. As regards license protection mode, one of the following dialog box will be displayed:

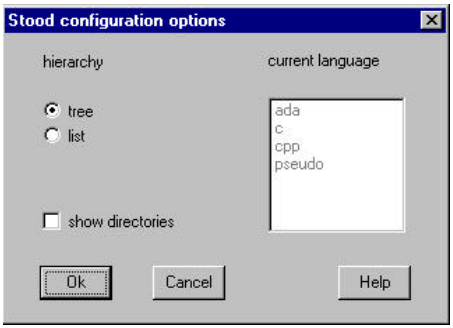
- hardware license key on **Windows** only:

- software time-limited key on **Unix** and **Windows**:



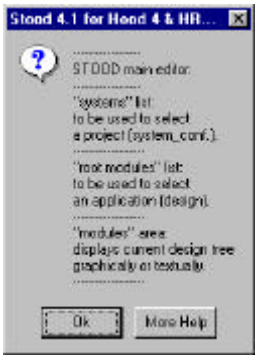
• sharable floating tokens license on an **Unix** or **Windows** server:

Options : This command may be used to set a few options, valid only during current session:



- *hierarchy* option may be set either to *tree* or *list* to control display mode of the **Design Tree** in right side area of *main editor*.
- *current language* option may be used to set or change default language for selected **Application**.
- *show directory* option enables full pathnames within *systems* and *root modules* lists.

Help : This command should be used to get general help about main editor:

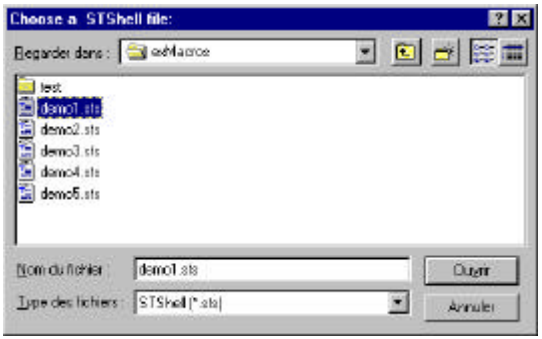


Contents of these dialog boxes may be customized by editing `main` and `main.more` help files. Refer to § 1.1.2.5 for further details about contextual help files

Duplicate : This command may be used to open another *main editor* within current session. This may be useful to open several **Applications** simultaneously.

Print : This command may be used to print current **Design Tree** on the default printer of your computer.

STShell : This command should be used to execute an **STShell** macro commands file interactively:

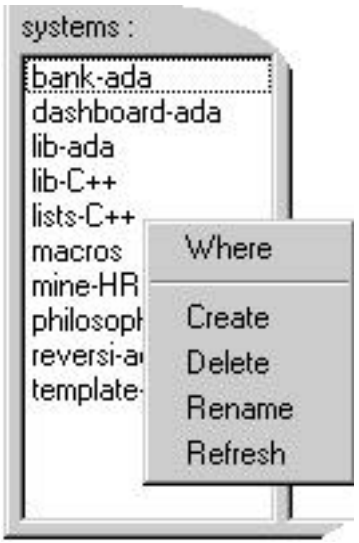


More informations about **STShell** commands may be found in § 1.3.1

Save : This commande must be used to save current **Application**.

Quit : This command should be used to quit current session. If current **Application** has been modified, a dialog box asks for saving modifications:

3.2. Project management



Left side list of *main editor* should be used to manage **Projects** (refer to § 2.1 to know more about **STOOD Projects**). This list shows all visible **Projects**, as specified by `SavePath` property in initialization file (`stood.ini` for **Windows** and `.stoodrc` for **Unix**). Refer to § 1.2.2 to know more about `SavePath` property. Only one **Project** may be selected at a time in a *main editor*. To use several **Projects** simultaneously, use *Duplicate* command in *Window* menu.

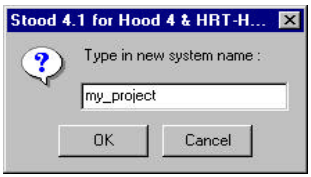
- To select a **Project**, click on chosen name in the list.
- To unselect current **Project**, click on highlighted name in the list.

A pop-up menu may be shown by pressing center or right menu button, while menu pointer is located within *systems* list borders. This menu provides a few **Project** related functions:

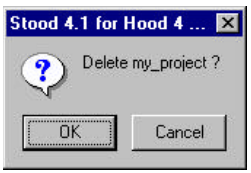
Where : Provides information about physical location of selected **Project**:

note: displayed information may be customized by modifying `infosyc.sh` internal tool (refer to § 1.1.2.4)

Create : This command should be used to create a new **Project**. **Project** name must be entered within a dedicated dialog box:



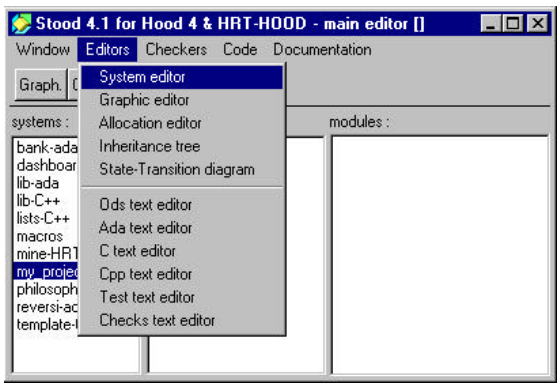
Delete : This command should be used to delete currently selected **Project**. Note that deleting a **Project** doesn't delete any **Application**. This action must be confirmed within a dedicated dialog box:



Rename : This command should be used to rename currently selected **Project**. It doesn't affect its contents. New name must be entered within a dedicated dialog box:

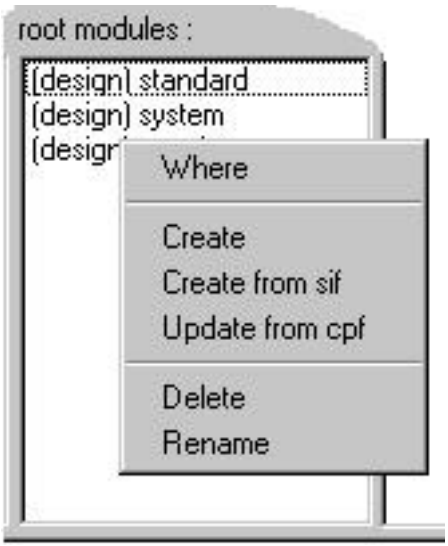
Refresh : This command should be used to refresh *systems* list contents during current session. This action can be useful when other active sessions have created or renamed **Projects**, or if **Projects** have been physically moved since current session has been opened.

A **Project** contains a list of references to **Applications**. To control this list, use *System editor* command of *Editors* menu:



This action opens a *system editor*. Please refer to chapter 4 to get detailed description about how to use a *system editor*.

3.3. Application management



Center list of *main editor* should be used to manage **Applications** (refer to § 2.2 to know more about **STOOD Applications**). This list shows all referenced **Applications** for currently selected **Project**, which may be accessed by searching `SavePath` property of initialization file (`stood.ini` for **Windows** and `.stoodrc` for **Unix**). Refer to § 1.2.2 to know more about `SavePath` property. Only one **Application** may be selected at a time in a *main editor*. To use several **Applications** simultaneously, use *Duplicate* command in *Window* menu.

- To select an **Application**, click on chosen name in the list.
- To unselect current **Application**, click on highlighted name in the list.

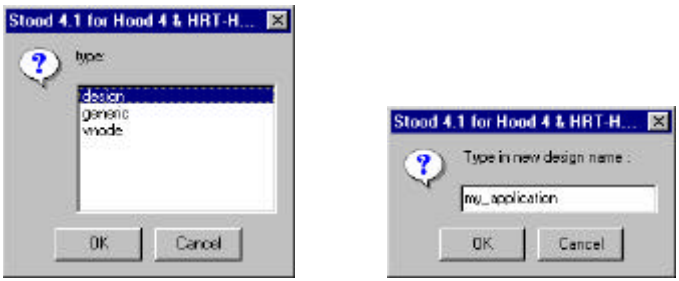
If an unselected **Application** was modified and not saved, following dialog box is displayed:

A pop-up menu may be shown by pressing center or right menu button, while menu pointer is located within *root modules* list borders. This menu provides a few **Application** related functions:

Where : Shows information about physical location of selected **Application**:

note: displayed information may be customized by modifying `inforoot.sh` internal tool (refer to § 1.1.2.4)

Create : This command should be used to create a new **Application**. After having defined the kind of **Application** to be created, **Application** name must be entered within a dedicated dialog box:



Create from sif : This command should be used to create a new **Application** from a file written with **SIF** syntax. Please refer to § 2.3.1 to get additional informations **S**tandard **I**nterchange **F**ormat.
entered in a dedialox box:

All `.sif` files found by searching `SavePath` are added to the list. To open another **SIF** file, use `<others>` choice, to open a standard file navigator.

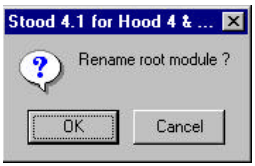
Update from cpf : This command should be used to update an existing **Application** by reading **CPF** instructions from a file. Please refer to § 2.3.2 to get additional informations about **C**hange **P**ropagation **F**ormat. Filename to be loaded must be entered in a dedicated dialox box:

All `.cpf` files found by searching `SavePath` are added to the list. To open another **CPF** file, use `<others>` choice, to open a standard file navigator. Note that an **Application** must be selected to use this function.

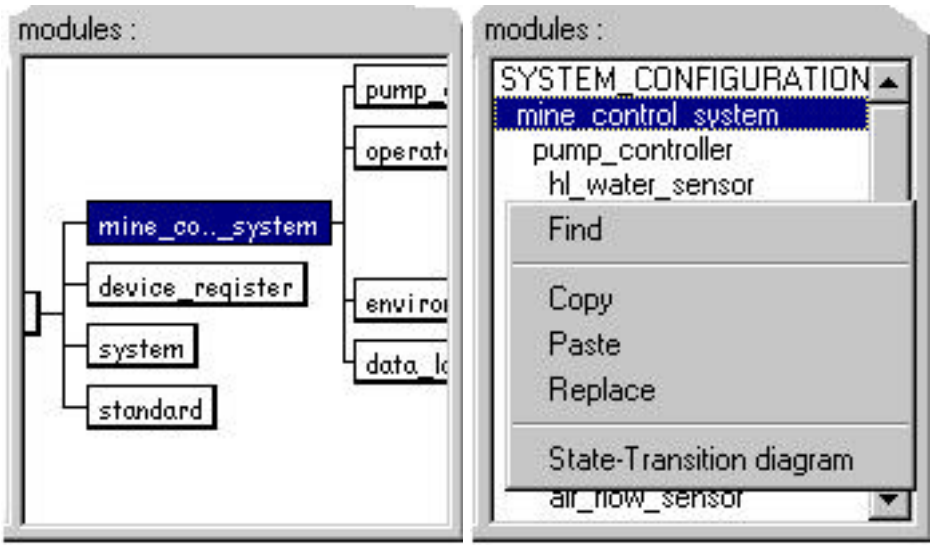
Delete : This command should be used to delete currently selected **Application**. This action must be confirmed within a dedicated dialog box:

IMPORTANT NOTE: A deleted **Application** cannot be recovered

Rename : This command should be used to rename currently selected **Application**. New name must be entered within a dedicated dialog box, and a confirmation is requested:



3.4. Module management



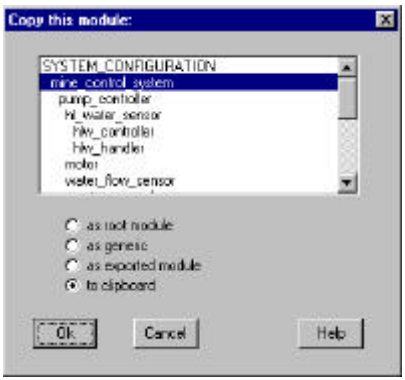
Right side area of *main editor* may be used to manage **Modules** (refer to part II of this documentation to know more about **STOOD Modules**). This area shows the **Modules** hierarchy for currently selected **Application**. This hierarchy is also called the **Design Tree**, and may be displayed either as a graphical tree (left above picture), either as a list (right above picture). This display mode is controled by *Options* command of *Window* menu (refer to § 3.1).

Please note that it is not possible to create nor delete **Modules** directly at this level. These actions may only be performed at architectural design level, with a *graphic editor* (refer to part II).

A pop-up menu may nevertheless be shown by pressing center or right menu button, while menu pointer is located within *modules* area borders. This menu provides a few **Modules** related functions:

Find : This command may be used to highlight currently selected **Module** in all open *graphic editors* and *text editors*.

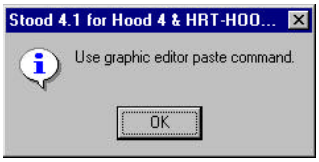
Copy : This command may be used to copy currently selected **Module** and its contents, to one of the proposed destinations:



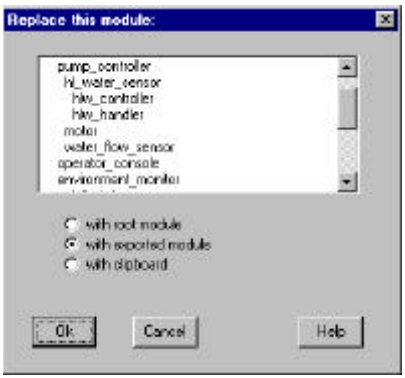
Possible destinations are:

- A newly created **Root Module** within current **Project**.
- A newly created **Generic Root Module** within current **Project**.
- A newly created exported **Root Module** within current **Project**.
- The shared temporary area (clipboard).

Paste : This action cannot be performed at this level, please use a *graphic editor* to paste a new **Module**:



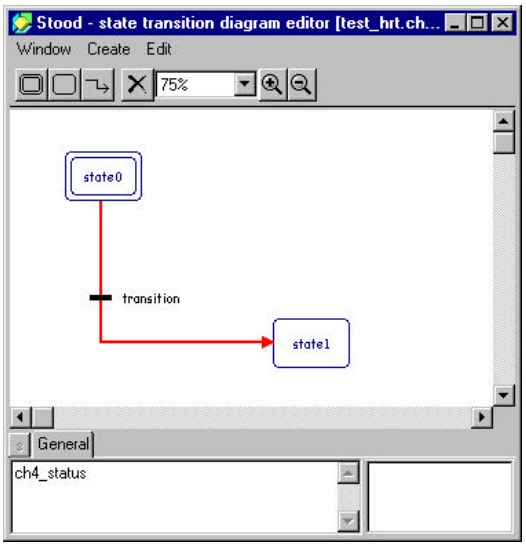
Replace : This command may be used to replace currently selected **Module** and its contents, by one of the proposed sources:



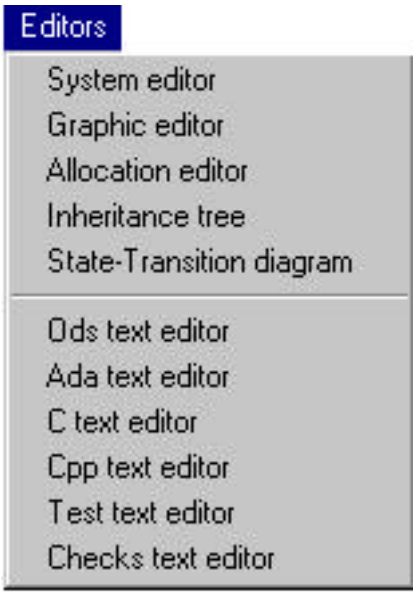
Possible sources are:

- An existing **Root Module** of current **Project**.
- An existing exported **Root Module** of current **Project**.
- The shared temporary area (clipboard).

State-Transition diagram : This command may be used to open a *state-transition diagram editor* for currently selected **Module**:



3.5. Graphical and textual editors



Menu *Editors* of *main editor* should be used to open **STOOD** editing windows. Higher part of the menu refers to *graphic editors* and cannot be customized, whereas lower part refers to *textual editors* which are defined in `DataBase` configuration file (refer to § 1.1.2.7). This menu may thus not look like the picture shown above.

System editor : Opens a *system editor* (refer to § 4), to manage current Project selected in systems list.

Graphic editor : Opens a *graphic editor* (refer to part II, § 3) to perform architectural design tasks on current Application selected in root modules list.

Allocation editor : Opens an *allocation editor* (refer to § 5) to perform a deployment on a distributed target. Current **Application** must be a **Virtual Node**.

Inheritance tree : Opens the *inheritance tree* (refer to part II, § 5).

State-Transition diagram : Opens a *state-transition diagram editor* (refer to part II, § 4) for current **Module** selected in modules area, or for current **Root Module** if no **Module** is selected.

Ods text editor : Opens an *ods text editor* (refer to part III, § 2) to perform detailed design tasks for current **Application**.

Ada text editor : Opens an *ada text editor* to display all **Ada** related parts of the **Application** (handly written and automatically generated code).

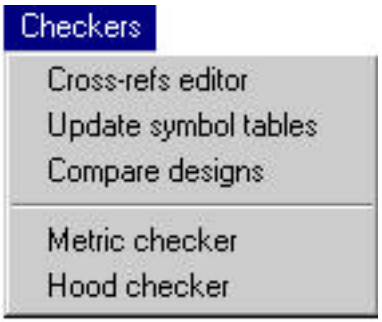
C text editor : Opens a *c text editor* to display all **C** related parts of the **Application** (handly written and automatically generated code).

Cpp text editor : Opens a *cpp text editor* to display all **C++** related parts of the **Application** (handly written and automatically generated code).

Test text editor : Opens a *test text editor* to manage unit testing of the **Application**.

Checks text editor : Opens a *checks text editor* to display information related to all *rules checkers* and *cross-reference tables*.

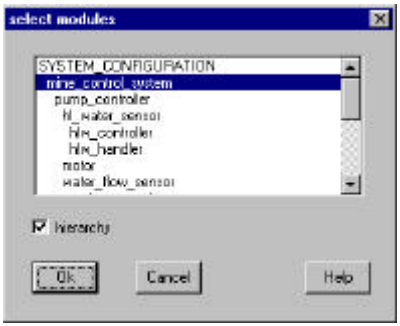
3.6. Checking tools



Menu *Checkers* of *main editor* should be used to open **STOOD** checking tools. Higher part of the menu refers to *cross-references tables* and *compare designs* tool, and cannot be customized, whereas lower part refers to *rules checkers* which are defined in `config/checkers` configuration directory (refer to § 1.1.2.3). As it is possible to configure **STOOD** in order to add or remove *rules checkers*, this menu may not look like the picture shown above.

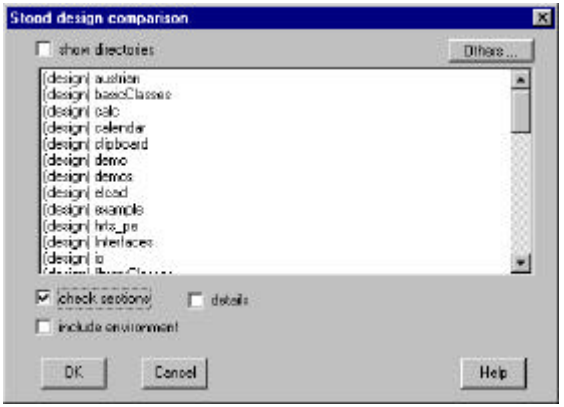
Cross-refs editor : Opens a cross-references table (refer to part III, § 4) for current target language.

Update symbol tables : This command may be used to perform a global update of all elementary symbol tables:



This action will be performed only for selected **Module** and its sub-hierarchy if relevant check box has been ticked off.

Compare designs : Opens a *compare designs* tool to compare current Application with a reference Application. This reference Application must be selected in the list:



Before launching design comparison, several options may be selected:

- *show directories* : enables display of a full pathname for **Applications**. This may be useful when several applications have the same name but are located in different directories.

- *check sections* : enables **ODS** sections comparison. It only checks whether the same sections are present or not

- *details* : when check sections has been ticked off, enables a full comparison of **ODS** sections contents.

- *include environments* : extends the scope of the comparison to the other **Root Modules** referenced by the **Application**.

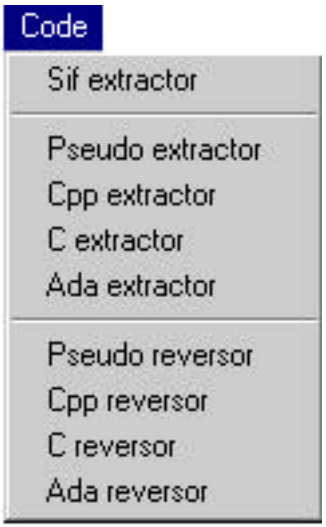
- *others* : opens a standard file navigator to select add another **Application** to the list. This may be useful to make a comparison with an **Application** not reachable by searching `SavePath` property.

Result of design comparison may be shown in a report file, which is automatically displayed in a *checks text editor*, when comparison is completed.

Metric checker: Opens a metrics calculation tool, to get statistical informations about **Application** size and complexity.

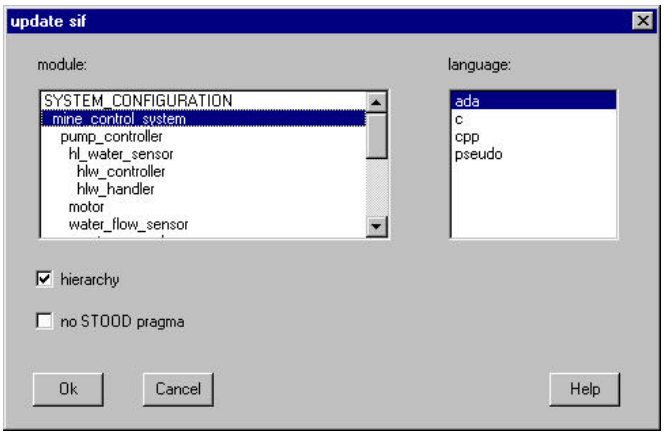
Hood checker: Opens a **HOOD** rules verification tool to check **Application** compliancy with standard design rules.

3.7. Code extractors and reversors



Menu *Code* of *main editor* should be used to open **STOOD** coding tools. This menu is composed of three parts. Higher part of the menu is dedicated to the **S**tandard **I**nterchange **F**ormat (**SIF**) generator, and cannot be customized. Middle part of this menu lists all *code generators* and lower part of the menu lists all reverse coding tools (*reversors*). There is a *code generator* and a *reversor* for each target language defined in `config/code_extractors` directory (refer to § 1.1.2.1). As it is possible to configure **STOOD** in order to add or remove *code extractors*, this menu may not look like the picture shown above.

Sif extractor : Opens a **SIF** generation tool:



Before launching a **SIF** generation, several options may be selected:

- *modules* : the **Module** for which **SIF** generation must be performed, should be selected in this list.

- *language* : **SIF** generation can be performed for a given target language only. Chosen language must be specified in this list.

- *hierarchy* : if **SIF** generation must be propagated to the sub-hierarchy of selected **Module**, this box should be ticked off.

- *no STOOD pragma* : **STOOD** specific information is inserted inside generated **SIF** file. To avoid that, please tick this box off.

As soon as these options have be selected, and *Ok* button has been pushed, a standard file navigator asks for the name of the **SIF** file to be generated

Pseudo extractor : In standard configuration, there is no *code generator* for **Pseudo Code**.

Cpp extractor : Opens a *code extractor* tool to generate **C++** source files from current **Application**.

C extractor : Opens a *code extractor* tool to generate **C** source files from current **Application**.

Ada extractor : Opens a *code extractor* tool to generate **Ada** source files from current **Application**.

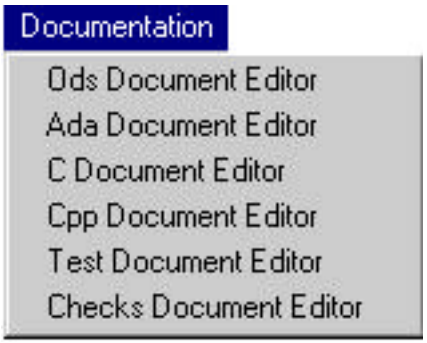
Pseudo reversor : In standard configuration, there is no *code reversor* for **Pseudo Code**.

Cpp reversor : Opens a *code reversor* tool to update current **Application** from **C++** source files.

C reversor : Opens a *code reversor* tool to update current **Application** from **C** source files.

Ada reversor : Opens a *code reversor* tool to update current **Application** from **Ada** source files.

3.8. Document generators



Menu *Documentation* of *main editor* should be used to open **STOOD** documentation tools and generate paper or electronic documentation from current **Application**. There is a *document editor* for each *text editor*. As it is possible to configure **STOOD** in order to add or remove *document editors*, this menu may not look like the picture shown above.

Detailed information about documentation tools (selecting sections to be printed, choosing an output format, ...) may be found in § 5 of part III of this manual.

With standard configuration, proposed documentation tools are:

Ods Document Editor : Opens a documentation tool to print a standard **HOOD** design document, from information that may be handled in *ods text editor*.

Ada Document Editor : Opens a documentation tool to print an **Ada** coding document, from information that may be handled in *ada text editor*.

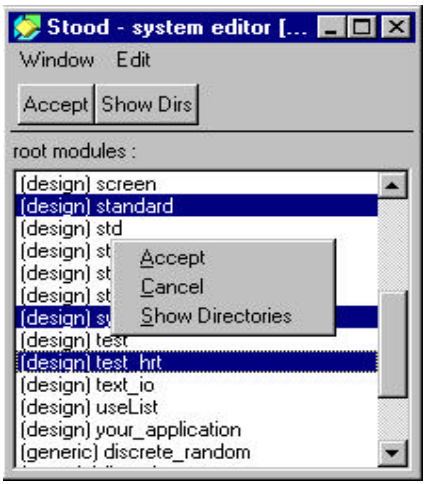
C Document Editor : Opens a documentation tool to print a **C** coding document, from information that may be handled in *c text editor*.

Cpp Document Editor : Opens a documentation tool to print a **C++** coding document, from information that may be handled in *cpp text editor*.

Test Document Editor : Opens a documentation tool to print an unit testing document, from information that may be handled in *test text editor*.

Checks Document Editor : Opens a documentation tool to print a check report document, from information that may be handled in *checks text editor*.

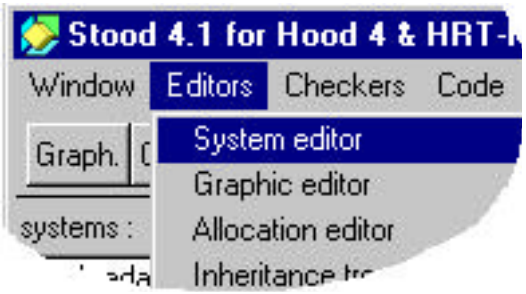
4. System editor



System editor may be used to specify the list of visible **Applications** within a given **Project** (refer to § 2 to know more about **STOOD Projects** and **Applications**). This window is composed of a scrollable list in which a multiple selection of items may be performed, a menu bar and a buttons bar. Buttons and contextual menu items of the list are only shortcuts for the commands provided by the *edit* menu

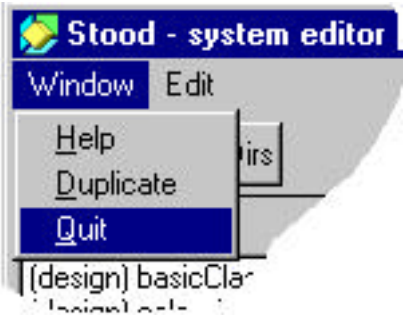
• To open a *system editor*:

Either select a **Project** in *systems* list of *main editor*, and choose *System editor* command in *Editors* menu of *main editor*, either simply double-click on a **Project** in *systems* list of *main editor*.

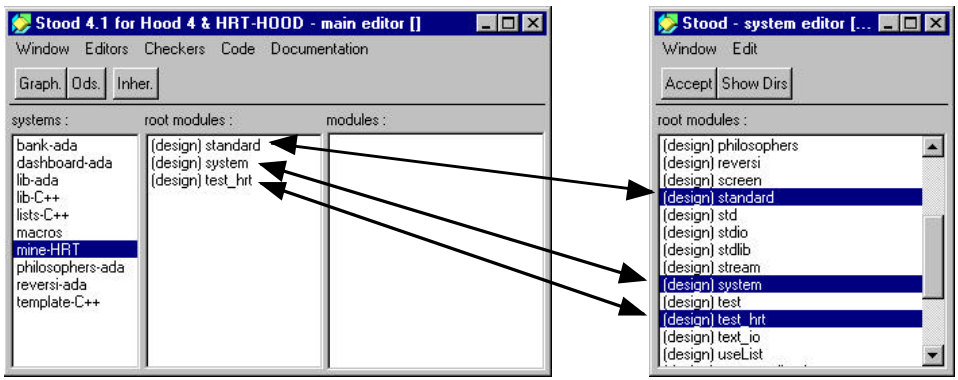


• To close a *system editor*:

Use *Quit* command in *Window* menu of appropriate *system editor*.



Items displayed in *root modules* list of *system editor* are obtained by a searching `SavePath` property defined in `stood.ini` or `.stoodrc` initialization file (refer to § 1.2.2 to know more about `SavePath` property).

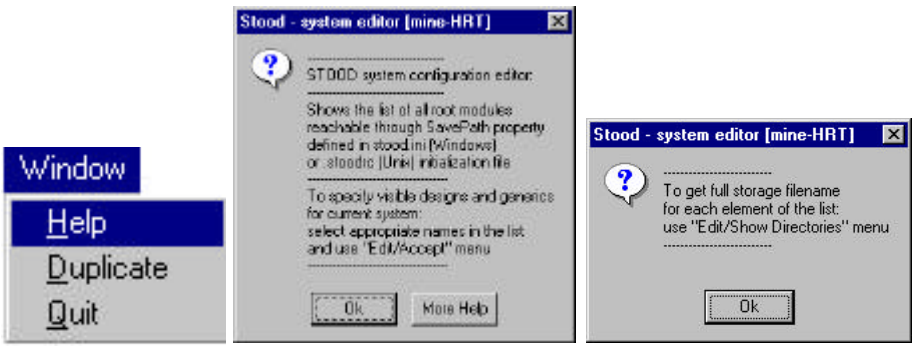


Highlighted items in *system editor* correspond to similar items in *main editor*. These

Project. To validate a new setting, *Accept* command in *Edit* menu of *system editor* must be used:

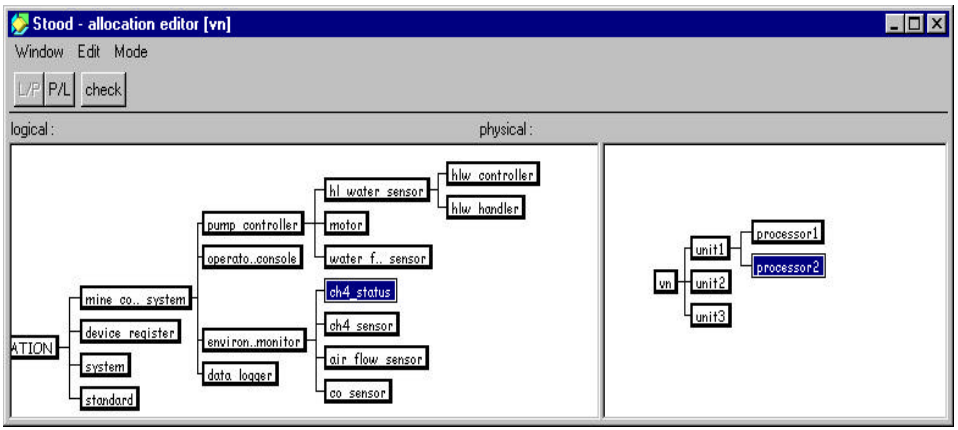
Simple name of a **Root Module** is sometimes insufficient to identify an **Application**. Two **Applications** of the same name may be located in different directories. To avoid errors, full pathname may be displayed in the list. To enable that feature, *Show Directories* command of *Edit* menu should be used.

Help about *system editor* may be displayed with *Help* command of *Window* menu:



Contents of these dialog boxes may be customized by editing `config/help/syc` and `config/help/syc.more` configuration files (refer to § 1.1.2.5 for further details about contextual help files).

5. Allocation editor



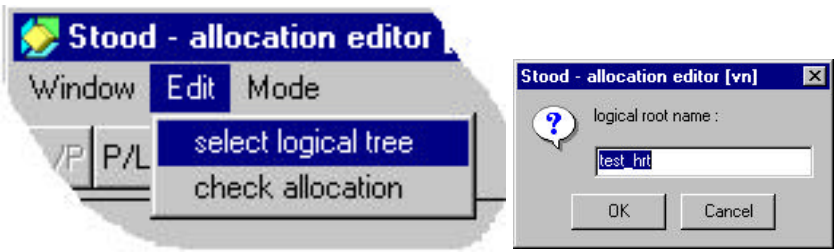
An *allocation editor* may be open only when current **Application** is a **Virtual Node**, that is a deployment model for a logical **Application** (a **Design**). The goal of allocation editor is to assist the user in building a distributed **Application**.

An *allocation editor* is composed of two graphic areas and a menu bar. Buttons may have been configured to act as shortcuts to menus commands. The two graphic areas, *logical* and *physical*, may be displayed either on the left either on the right side of the window. To control this swap, use *Mode* menu. Left side area only accepts simple selection, whereas right side area accepts multiple selections.

An allocation operation should follow several steps:

- load a logical **Application**:

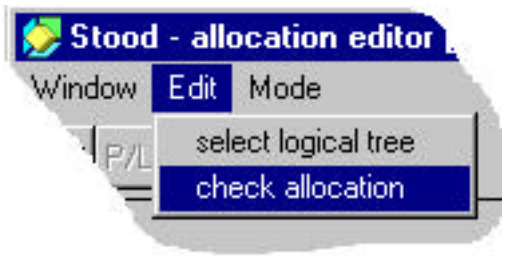
To load a Design, use select logical tree command in Edit menu, and enter the name of an existing Design:



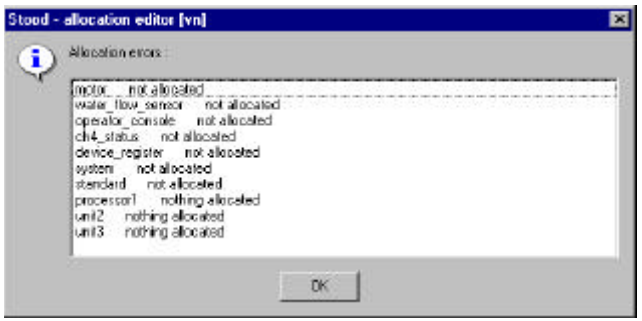
- select at least one allocation node for each leaf or branch of the logical **Application**:

Two modes are available to perform this action. In first mode (*L/P*), left side area shows the logical tree, and the right side area shows the physical tree, and each
In the other mode (*P/L*), left side area shows the physical tree, and the right side area shows the logical tree, and each selected physical element may encompass one or more logical elements.

• check allocation completeness.



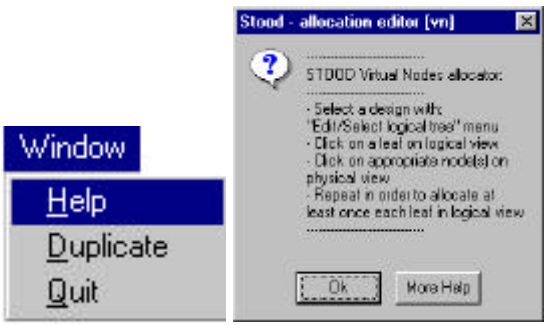
Allocation completeness may be verified with *check allocation* command of *Edit* menu. A error report is then displayed in a dialog box:



Allocation information may be used by code generators to produce distributed software. For instance, mapping to **Ada95** partitions is quite immediate. It should be noticed that the main basic concepts of the **HOOD** design process highly facilitate the development of distributed **Applications**:

• Modularity makes partionning easier
• Minimization of coupling optimizes communication channels

Help about *allocation editor* may be displayed with *Help* command of *Window* menu:



Contents of these dialog boxes may be customized by editing `config/help/vna` and `config/help/vna.more` configuration files (refer to § 1.1.2.5 for further details about contextual help files).