Stood

Administrator Manual

revision A

Pierre Dissaux

This manual explains how to customize **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 the *Installation Manual* in case of any problem. **Stood v5** is available for both **Unix/Motif** and **Windows** platforms.

1. Administrator’s customizations

This chapter contains usefull information to check and customize current installation of the product on your system. Following components should be found after a standard installation of **Stood v5** on your system:

1.1. Binary files

1.1.1. Supported platforms

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

- `sol2` for **Solaris2** on Sun **sparc** platforms.
- `w32` for **Windows** on **PC** platforms.
- `pclinux` for **Linux/OpenMotif** on **PC** platforms.

Binaries for other platforms may be available. Please contact **TNI**’s technical support for further informations: `stood@tni-world.com`

Please note that **VMS** on **DEC Vax** and **Alpha** as well as **AIX** on **IBM RS6000** and **hpux** on **HP700** are no more supported.

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
`crev`	C syntactic analyser
`aadlrev`	AADL 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 the `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 and to specify an appropriate working directory in its startup property.

1.1.3. Ancillary files

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

`stood.eng`	Stood localization file
`stood.bmp`	optional alternate startup image

1.1.4. Initialization file

The `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 §2.

If no other initialization file is found, **Stood** will use the one located inside the `bin` directory. Initialization file gathers all user's level customizations. It is recommended not to modify the initialization file provided during a standard installation procedure of the product. However, customized initialization files may be created in the various working or home directories. These additional initialization files just need to contain the actually overloaded properties. All the other properties will be set to their default value, as defined in the `bin` directory.

Many other customization capabilities are available at administrator's level. These other customizable features are located inside `config` directory and will be listed later in this chapter.

1.2. Configuration files

The `config` directory is the general container for all the platform independant configuration files, including the documentation and code generators. Features contained in this directory may all be customized by the administrator of the tool. Many of these features operate like plugins, that may be added, modified or removed with a simple "plug and play" maintenance process.

The standard configuration complies as far as possible with the **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**, and other standards like the Architectural Analysis and Design Language (**AADL**) or the version 2.0 of the Unified.Modeling Language (**UML**) are now supported in the last versions of the configuration.

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 specific code generation, documentation and verification rules;
- implement communication 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 §2.6 for further details.

1.2.1. Code generators

The code generators are located inside the `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++**
- `config/code_extractors/aadl` **AADL**

Each of these subdirectories contains a set of files that are used by **Stood** each time corresponding code generation action is invoqued. The code generation rules are written in **prolog** language. When starting code generation, **Stood** produces a **prolog** facts base and gives the control to a **prolog** engine which loads both facts and rules bases, to generate source code files (refer to §1.4).

Contents of a code generation directory is shown below. Some of these files may be customized by the 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)
`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 the user's manual. Like the other the plugins, the code generators may be updated more frequently than the **Stood** kernel. To know the precise version of a code generator, edit the `Extract.pro` file, which header provides the date of the last modifications.

The file `extractors` is used to define different profiles for the code generation process (for instance, full or partial generation), and to specify the configuration variables that can be used to handle the source code suffix:

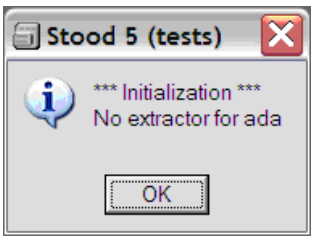
```
SuffixSpecVariable        "ADASPECSUFFIX"
```

defines an initialization file property in the `Languages` category(cf. §2.3.2).

```
ProcedureSpecVariable     "$As"
```

defines the corresponding pseudo variable for the DataBase file (cf. §1.2.9).

While launching **Stood**, a set of verifications are performed on the configuration files. It may occur that, due to specific customizations, some unconsistencies appear. The main issues may come from differences between the `DataBase` file and the various configuration subdirectories it refers to. This is especially important for the code generators that can easily be removed or added to the configuration directory, whereas the `DataBase` file has not necessarily be properly updated. For instance, if the **Ada** code generator has been removed from the configuration directory, but some `DataBase` sections still refer to the **Ada** language, then the following warning will be shown at launch time:



1.2.2. Document generators

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

- `config/doc_extractors/html` **HTML** file
- `config/doc_extractors/tps` **Interleaf** input file
- `config/doc_extractors/mif` **FrameMaker** input file
- `config/doc_extractors/ps` **PostScript** file
- `config/doc_extractors/rtf` **MSWord** input file
- `config/doc_extractors/pdf` **PDF** file

Each of these subdirectories contains a set of files that are used by **Stood** each time corresponding document generation is invoqued.

Document generators are written either in **prolog** language (those whose directory contain a `prolog` file), either using a the **docBook** technology (those whose directory contain a `docbook` file). Note that the **easyDoc** technology is no more supported. Both kinds of generators may be customized by the 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 graphics (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
`gif`	insertion of GIF graphics

Important notes:

- The file `printer.sh` is used to send produced document to a printer or a documentation tool. The tool administrator must customize there the actual name of the printer or print spooler to use.
- The file `header.xxx` (where `xxx` may be `ps`, `tps` or `mif`), may be edited to change the fonts to be used in the documentation.
- Other `.sh` files may be created to propose different printing modes or different printers to the user. When `print.sh` is the only one defined, only the *file only* entry is proposed in the *tools* menu. When additional scripts are defined, corresponding entries are automatically proposed in the *tools* menu.

In addition to `.cfg` files, **docBook** document generators uses the `doc_templates` subdirectory that contains a customizable template for each corresponding format.

`config/doc_templates/portrait.rtf`	template for Word documents
`config/doc_templates/portrait.style`	template for PDF documents

These files may be edited to customize the layout of the documents that are generated by **Stood** for these formats.

1.2.3. Rules checkers

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

- `config/checkers/hood` design rules checker
- `config/checkers/metric` design metrics
- `config/checkers/requirements` requirements coverage
- `config/checkers/scheduling` schedulability analysis
- `config/checkers/database` design storage checker
- `config/checkers/aadl` **AADL** rules checker

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

Contents of a rules checker directory is shown below. Some of these files may be customized by the 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
`config`	to specify the additional languages to process

These files are dupplicated for each checker plugin. In addition to these files, a set of specific files are contained inside each checker subdirectory. The `.pro` suffix identifies the prolog source files and the `.sbp` suffix is used for the corresponding prolog binary files. Only the binary file is required in normal use. The source file is necessary to perform customizations in the default rules.

Specific files for the design rules checker:

`General.pro (.sbp)`	general design 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

Specific files for the metric rules checker:

`Cohesion.pro (.sbp)`	cohesion of the design
`Coupling.pro (.sbp)`	coupling between design entities
`Depth.pro (.sbp)`	depth of the design hierarchy
`Size.pro (.sbp)`	size of the design entities
`Other.pro (.sbp)`	miscellaneous metrics

Specific files for the requirements coverage checker:

`Coverage.pro (.sbp)`	summary of coverage information
`ERMatrix.pro (.sbp)`	entity -> requirements matrix
`REMatrix.pro (.sbp)`	requirement -> entities matrix
`TabMatrix.pro (.sbp)`	tabulated text matrix
`Output.pro (.sbp)`	output file for Reqtify

Specific files for the schedulability analysis checker:

`RMA.pro (.sbp)`	schedulability analysis

Specific files for the design database checker:

`Hierarchy.pro (.sbp)`	hierarchy of components
`Scan.pro (.sbp)`	list of files in the database

Specific files for the **AADL** rules checker:

`Legality.pro (.sbp)`	AADL legality rules

More details about contents and use of these files is provided in the users's documentation. Like other the plugins, the rules checkers may be updated more frequently than the **Stood** kernel.

1.2.4. Tools

Even for **Windows** platforms, **Stood** uses **Unix** shell scripts to control the interface between the kernel and the post-processors or the configuration/version management system, or more simply the file storage environment, and to easily call remote tools. These scripts may all be customized by the tool administrator, if required. For safety reasons, they are stored into two different configuration subdirectories: `internalTools` and `externalTools`. Internal tools should never be removed as they are used as gateways between the kernel and the post-processors (rules checkers, code and document generators) and the configuration/version management system or the file system. Contents of the `config/internalTools` configuration subdirectory is as follow:

`lock.sh`	called when openning a session
`inittrash.sh`	called when closing a session
`infosyc.sh`	called when inquiring about a System
`inforoot.sh`	called when inquiring about a Design
`copydir.sh`	called when copying or moving files
`rmdir.sh`	called when deleting files
`fastprint.sh`	called for direct printing of graphics and trees
`print.sh`	called to call the document generator
`scan.sh`	called when accepting source code
`external.sh`	called to launch checkers and extractors
`difffiles.sh`	called when comparing files
`reqtify.sh`	called when importing requirements from Reqtify
`reverse.sh`	called to launch a reverse engineering engine
`copyenv.sh`	called when copying the interface of an Environme
`checkin.sh`	called for configuration management checkin (*)
`checkout.sh`	called for configuration management checkout (*)
`checklock.sh`	called for configuration management lock (*)
`checkunlock.sh`	called for configuration management unlock (*)

(*) `checkin.sh`, `checkout.sh`, `checklock.sh` and `checkunlock.sh` are generic names. Several sets of scripts may be defined to interface with various configuration or version management systems.

The name of the actual scripts that will be used, are specified by the value of `CheckInProcedure`, `CheckOutProcedure`, `CheckLockProcedure` and `CheckUnlockProcedure` properties in the initialization file (`stood.ini` for **Windows** or `.stoodrc` for **Unix**). Standard installation provides an interface with **cvs** (`cvsin.sh`, `cvsout.sh`, `cvslock.sh`, and `cvsunlock.sh`) and a simple version management system operating by copy (`copyin.sh`, `copyout.sh`, `copylock.sh`, `copyunlock.sh`).

The `CheckOutProcedure` is used to extract a given version from the configuration management database to the local working area. The `CheckInProcedure` is used to save the current contents of the local working area into a given version in the configuration management database. The `CheckLockProcedure` and `CheckUnlockProcedure` can be used to manage multiple access to the configuration management database.

When called by **Stood**, these configuration management procedures receive the following parameters:

`$1`	name of the application to load, lock, unlock or save.
`$2`	base directory in the local working area.
`$3`	version id (versions ids are defined in the initialization file).
`$4`	selected module (not currently used)
`$5`	protection flag (ro for read-only or rw for read-write)
`$6`	filename with the list of components to be processed

The last parameter is present if the property `CheckInWithArgFile` or `CheckOutWithArgFile` is set to `Yes`.

Unlike the internal tools, the use of external tools is optional. 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)
`check_aadl.sh`	launch the AADL syntactic code analyser
`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`	current Property file pathname (if any)
`$2`	current Design name
`$3`	current Component name
`$4`	current Feature name
`$5`	current Property identifier (logical name)

For instance, the external tool `info.sh` is defined as follow:

```
#!/bin/sh
echo "design name: " $2
echo "component name: " $3
echo "feature name: " $4
echo "property logical name: " $5
echo "property file pathname: " "$1"
```

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 tool suspends **Stood** until its completion, except if the header of the relevant script contains the following statement:

```
#!/bin/sh
# Stood:NoWait
```

In that case, the script mustn't write anything on the standard output.

1.2.5. Contextual help files

A on-line help mecanism is available with **Stood**. It is also fully customizable by the tool administrator. Help facility is composed of three different parts, each of them being 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 or view identifiers also used for setting initialization file properties (refer to §2.4) or in the **STShell** language parameters (refer to §3.1.1).

`main`	main window
`tre`	design tree
`hie`	inheritance tree
`req`	requirements editor
`gra_hood`	HOOD graphical editor
`gra_uml`	UML graphical editor
`txt`	text editor
`std`	states-transitions diagram (STD) editor
`chk`	design verification tools
`utr`	call tree and access graph
`ext`	code extractor
`code`	code editor
`rev`	code reversor
`doc`	document editor
`vna`	allocation editor
`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
`dbconf`	configuration management dialog box

Additional help is provided for each section of the detailed design structure. This is particularly useful to give proper advices about the best way to insert information inside each section of the **ODS** (Object Description Skeleton). These advices may be informative text or examples of text input that are directly inserted at the right place. Both may be customized by the tool administrator, by editing files contained in the `ods_help` and `ods_template` configuration subdirectories.

Organization of these two subdirectories is directly related to the way **Application** storage has been configured in the `DataBase` file (refer to §1.2.9). Help and template information files are organized as any **Stood Application**, but in a generic way. Each time an **Application**, **Component** or **Feature** name is required to build an actual storage pathname, the reserved keyword `name` is used instead.

It is also possible to provide information for sections that are not stored in a file, but extracted from the design model by a procedure. In this case, help and template files will be named `proc#`, where # is the procedure number defined inside the `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 the following contextual pseudo-variable:

`$Dg`	Application name	`$St`	current config directory
`$Op`	Operation name	`$39`	Super-Class (Ada syntax)
`$Os`	Operation-Set name	`$40`	Super-Class (C++ syntax)
`$Ty`	Type name	`$70`	Attributes (Ada syntax)
`$Cp`	Constant name	`$71`	Attributes (C/C++ syntax)
`$Ex`	Exception name	`$72`	Operation Parameter list
`$Da`	Data name	`$73`	Enumeration (C/C++)
`$Id`	RCS tag	`$74`	Enumeration (Ada syntax)
`$Ho`	current SavePath directory		

Other pseudo-variables may be defined in the initialization file. For instance, to use a pseudo-variable `$Pj` which gives the name of the project, first declare it as follow in the `stood.ini` file:

```
[Variables]
Pj=Stood
```

or in the `.stoodrc` file:

```
Variables.Pj:Stood
```

Immediate contents of `ods_help` and `ods_template` configuration subdirectories refer to the first directory level (sections global to an **Application**):

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

`proc1`	list of child Components	*Read Only*
`proc2`	contents of current System Configuration	*Read Only*
`proc3`	list of requirements	*Read Only*
`proc4`	DataFlows	*Read Only*
`proc5`	Exception Flows	*Read Only*
`proc15`	actual parameters for Instance_Of Generic Components	
`proc16`	instance range for Instance_Of Generic Components	
`proc22`	Operation declaration	
`proc23`	Used Operations	*Read Only*
`proc24`	Operation properties	
`proc29`	Type properties	
`proc30`	Class inheritance	
`proc31`	Type attributes	
`proc32`	Exception definition	
`proc33`	propagated Exceptions	
`proc34`	Constrained Operations	
`proc35`	OBCS is Implemented By	*Read Only*
`proc36`	Required Interface (Op, Os, Ty, Co, Ex)	*Read Only*
`proc37`	Operation Set definition	*Read Only*
`proc38`	Type enumeration	
`proc46`	Required Interface (Op)	*Read Only*
`proc47`	Required Interface (Op, Ty, Co, Ex, Da))	*Read Only*

(cont. next page)

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*
proc175	sketch editor	
proc176	table editor	
proc199	inheritance tree	*Read Only*
proc200	Design tree	*Read Only*
proc220	State-Transition diagram	
proc224	Transition event	*Read Only*
proc225	State exiting Transitions	*Read Only*
proc226	State entering Transitions	*Read Only*
proc227	origin State for the Transition	*Read Only*
proc228	destination State for the Transition	*Read Only*
proc302	Object Real-Time Attributes	
proc303	Operation Real-Time Attributes	

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

- `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
`modif.t`	help and template files for Component modifications
`header`	help and template files for code files header
`ProDes.t`	help and template files for Project Description

- `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.t`	help and template for Operation changes file
`name_header.u`	help and template for Op. Ada separate file header

- `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

- `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

- `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

- files:

`PRAGMA`	help and template files for Component 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

1.2.6. Icons

Stood uses customizable icons, especially when displaying buttons or menu items. Icons definition files are stored in `icons` configuration subdirectory. Each icon is described by a pair of `*.bmp` and a `*M.bmp` files for both **Windows** and **Unix** platforms. It is possible to edit these files with an appropriate utility program to change the icons, or add other icons and associate them to some window buttons bars (refer to §2.4.2). The default list of icons that are available in the `config/icons` configuration subdirectory is shown in the table below:

- icons for the Component and Property lists

 `lock.bmp`

 `partially lock.bmp`

 `write.bmp`

 `rubempty.bmp`

 `rubpartiallyfull.bmp`

 `rubfull.bmp`

 `save small.bmp`

 `print small.bmp`

 `print not small.bmp`

 `contains find text.bmp`

 `find text.bmp`

 `contains rev.bmp`

 `enable rev.bmp`

 `disable rev.bmp`

• icons for the requirements view (req)

```
load requirements from reqtify.bmp
load requirements from text.bmp or load.bmp
export requirements.bmp or save.bmp
new requirement.bmp
copy reference.bmp or copy.bmp
delete requirement.bmp
update coverage.bmp
help.bmp
```

• icons for the textual edition view (txt)

```
textpad.bmp
emacs.bmp
help.bmp
```

• icons for the UML graphical view (gra_uml)

```
print.bmp
new uml component.bmp
new uml class.bmp
new uml cyclic component.bmp
new uml sporadic component.bmp
new uml protected component.bmp
new uml feature.bmp
new uml assembly.bmp
new uml delegate provided.bmp
new uml inheritance.bmp
new uml aggregation.bmp
new uml delegate required.bmp
state diagram or state-transition.bmp
zoom in.bmp or loupe_p.bmp
loupe_e.bmp
zoom out.bmp or loupe_m.bmp
help.bmp
```

• icons for the HOOD graphical view (gra_hood)

```
print.bmp
show operations.bmp
show types.bmp
show constants.bmp
show exceptions.bmp
show data.bmp
new object.bmp
new class.bmp
new cyclic.bmp
new sporadic.bmp
new protected.bmp
new feature.bmp
new set.bmp
new use connection.bmp
new implementedby connection.bmp
new inheritance.bmp
new aggregation.bmp
state diagram.bmp or state-transition.bmp
zoom in.bmp or loupe_p.bmp
loupe_e.bmp
zoom out.bmp or loupe_m.bmp
help.bmp
```

• icons for the design verification view (chk)

```
update cross ref.bmp or update xref.bmp
find.bmp
call tree.bmp
access tree.bmp
check design.bmp
check system.bmp
aadl.bmp
ada.bmp
c.bmp
cpp.bmp
pseudo.bmp
help.bmp
```

• icons for the code extractors view (ext)

```
add pragma.bmp or pragmas.bmp
full extraction.bmp
obcs extraction.bmp
body only.bmp
help.bmp
```

• icons for the code editors view (code)

```
reverse.bmp
help.bmp
check ada.bmp
check aadl.bmp
```

• icons for the code reversor view (rev)

```
previous change.bmp
next change.bmp
update.bmp
update all.bmp
help.bmp
```

• icons for the documentation view (doc)

```
apply.bmp
select all.bmp
apply to all.bmp
html.bmp
mif.bmp
pdf.bmp
ps.bmp
rtf.bmp
tps.bmp
print.bmp
help.bmp
```

- icons for the deployment view (vna)

```
select design.bmp or logical root.bmp
check allocation.bmp
help.bmp
```

- icons for the state transtion diagram

```
print.bmp
istateuml.bmp
state.bmp
connect.bmp
event.bmp
delete2.bmp
undelete.bmp
loupe_p.bmp
loupe_m.bmp
help.bmp
```

- icons for the call, access, design and inheritance trees

```
print.bmp
tree.bmp
list.bmp
help.bmp
```

- icons for the sketch editor

The icons shown on the buttons bar of the sketch editor cannot be customized.

1.2.7. Http

Stood commands may be invoqued on an intranet or the internet. For **Stood** to operate as an **http** server, the `HttpServerPort` property must be properly set in the initialization file. If so, a connection to **Stood** may be established from any http client navigator with the following **URL**:
`http://host:port`

Where:

- `host`: must be a hostname where **Stood** runs.
- `port`: is the value define by the `HttpServerPort` property.

Example:
`http://server.tni-world.com:80`

When the connection has been established, the list of *STShell* scripts located in the `config/http` configuration subdirectory may be executed. For further information about *STShell*, please refer to §3.1.

Note that **http** is not the only protocol that may be used to send *STShell* commands to **Stood**. A **DDE** port is made available automatically on **Windows** and **Unix**, and a named pipe, named `st`, is automatically created on **Unix** only (if not disabled by the `Server.DisableSTShellPipe` property). Finally, a *STShell* filename may be specified on the command line when launching **Stood**.

1.2.8. Reverse

Stood now includes full reverse engineering features for **Ada**, **C**, and **AADL** source files. This reverse engineering process operates in three sequential steps. Firstly, an appropriate syntactic analyser is used to parse the source files. Then the semantic transformation is performed by a program written in prolog language to produce a **SIF** (Standard Interchange Format) file. Finally, **Stood** imports the **SIF** file to build the design. The `config/reverse` configuration subdirectory contains all the files required by the second step (**SIF** generator from **Ada**, **C** or **AADL** source files). There is a dedicated subdirectory for each installed reverse engineering tool:

- `config/reverse/ada` **Ada**
- `config/reverse/c` **C**
- `config/reverse/aadl` **AADL**

Each of these subdirectories contains a set of files that are used by **Stood** each time the corresponding reverse action is invoqued. The reverse rules are written in **prolog** language. When starting a reverse operation, **Stood** produces a **prolog** facts base and gives the control to a **prolog** engine which loads both facts and rules bases, to generate a **SIF** file (refer to §1.4).Contents of a reverse directory is shown below. Some of these files may be customized by the 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 reverse engineering (rules base)
`go.sh`	launching shell script
`makefile`	to re-build reverse rules if required

1.2.9. DataBase file

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

- Add or remove sections in the standard detailed design structure
- Add sections for new target languages (**Fortran**, **Java**, ...)
- Create or customize the textual editors
- Change the standard documentation layout
- Modify the **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
| cy | sp | pr

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

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

a	Active Component
o	Op_Control Component
i	Instance_Of Component
f	Formal_Parameters Component
e	unbound Environment Component
c	Class Component
sroot	System_Configuration
root2	bound Environment Component
root	Root_Component
leaf	Terminal Component
constr	Component providing at least one Constrained Operation
sif	to specify that this section should not appear in SIF files
cy	Hard Real Time Cyclic Object
sp	Hard Real Time Sporadic Object
pr	Hard Real Time Protected Object

These conditions may be combined using following 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 Component
`$Op`	name of current Operation (if relevant)
`$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)

According to the standard definition of the extractors file in the code_extractors directories (cf. §1.2.1), the following additional pseudo variables may be used to handle source code suffix:

`$As`	suffix for Ada specification files
`$Ab`	suffix for Ada body files
`$Cs`	suffix for C header files
`$Cb`	suffix for C source files
`$Ks`	suffix for C++ header files
`$Kb`	suffix for C++ source files
`$Ps`	suffix for pseudo code specification files
`$Pb`	suffix for pseudo code body files
`$Dl`	suffix for AADL files

Finally, other pseudo variables may be defined in the initialization file, by creating new entries in the `Variables` category. For instance, if the following lines are added to the `stood.ini` file:

```
[Variables]
Pj=Stood
```

Then, the `$Pj` pseudo variable can be used in the `DataBase` descriptor file. It will be replaced by its value in every pathname where it is used.

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

1	list of child Components	21	OPCS end
2	contents of System Config.	22	Operation declaration
3	List of Requirements	23	Used Operations
4	DataFlows	24	Operation properties
5	Exception Flows	29	Type properties
15	parameters for Instance_Of	30	Class inheritance
16	instance range for Instance_Of	31	Type attributes
17	begin of ODS	32	Exception definition
18	type of current Component	33	propagated Exceptions
19	end of ODS	34	Constrained Operations
20	OPCS begin	35	OBCS is Implemented By

(cont. next page)

36	Required Interface	91	symbol is used by
37	Operation Set definition	92	symbol name
38	Type enumeration	93	symbol uses
39	Superclass (Ada syntax)	94	Call Tree
40	Superclass (C++ syntax)	95	Inverse Call tree
41	child Operation	175	sketch
42	child Type	176	table
43	child Constant	199	Inheritance Tree
44	child Exception	200	Design Tree
45	child Data	201	Operations Diagram
46	Required Interface (Op only)	202	Types Diagram
47	Required Interface (with Data)	203	Constants Diagram
51	Operation name	204	Exceptions Diagram
52	Operation Set name	205	Data Diagram
53	Type name	206	UML Components Diagram
54	Constant name	211	Parent Operations Diagram
55	Exception name	212	Parent Types Diagram
56	Data name	213	Parent Constants Diagram
61	Operation is Implemented By	214	Parent Exception Diagram
62	Type is Implemented By	215	Parent Data Diagram
63	Constant is Implemented By	216	Parent UML Diagram
64	Exception is Implemented By	220	STD
65	Data is Implemented By	221	Parent STD
66	Operation Set Implemented By	222	State name
70	Type Attributes (Ada syntax)	223	Transition name
71	Type Attributes (C/C++ syntax)	224	Transition event
72	Operation signature	225	State exiting Transitions
73	Type enumeration (C/C++ synta:	226	State entering Transitions
74	Type enumeration (Ada syntax)	227	Transition origin State
81	Operation Set contents	228	Transition destination State
82	Type Set contents	301	current version
83	Constant Set contents	302	Component Real-Time Attribute
84	Exception Set contents	303	Operation Real-Time Attributes
85	DataSet contents		

```
(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** | **1**X[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 **Feature** 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
97	list of Tests
1XYZ	list of Operations, Types, Constants, Exceptions and Data

In the latter 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	
0	Provided
1	Internal
2	both

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

(19)DocType ::= TXT | CODE | TXTEND
| POSTSCRIPT | TABLE
```

A specific documentation procedure may be applied to a section. They must be processed by each document generator. Default procedures are:

TEXT	plain text
CODE	fixed font text
TEXTEND	plain text without form feed
POSTSCRIPT	formated graphics (EPSF, GIF, WMF, ...)
TABLE	Spreadsheet

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

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

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

With the `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*
`eAADL`	*aadl text editor*
`eChecks`	*checks text editor*
`eTests`	*tests text editor*

To create a new text editor, the 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 in the *editors* menu of the *main editor*.

The last keyword may be used for special needs:

The **`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 Component** will point to the contents of regarding section in relevant **Terminal Component**, if **Implemented_By** relationship have been properly set.

Example of DataBase sections:

```
'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 **Component**, except bounded **Environments** and **Formal Parameters** of **Generics**. 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 **Component**. It is visible in *ods text editor*, *ada text editor*, *c text editor* and *cpp text editor*.

1.3. Applications examples

Stood standard installation contains a set of directories with a few **Application** examples that may differ from a distribution to another. A typical distribution contains:

- `examples`: a few **AADL**, **Ada**, **C** and **C++** examples
- `libs`: interfaces to libraries (**AADL**, **Ada**, **C**, **C++**)
- `tutorial`: a set of executable demonstrative scripts

To execute the tutorial scripts on a **Windows** platform, simply double-click on the `lesson*.sts` icons located in the tutorial directory. Start with the `lesson1.sts` to create a new **Application**, then follow the instructions.

To execute the tutorial scripts on a Unix platform, launch the following command lines from a terminal:

```
stood -batch -f lesson1.sts
stood -batch -f lesson2.sts
```

...

1.4. Prolog engine

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 with the standard distribution in order to let the tool administrator use another **prolog** environment, if needed.

Stood doesn't require the source files of the **prolog** engine and libraries to work properly. They may thus be removed from the **Stood** execution environment. However, the `sbprolog` directory should contain at least:

`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

The executable file for the **prolog** engine is located into the `bin.xxx` directory. **Stood** always launches the **prolog** engine through **Unix** shell scripts:

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

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

```
SIMPATH=
    "$STOODPRO/modlib":
    "$STOODPRO/lib":
    "$STOODPRO/cmplib"
export SIMPATH
```

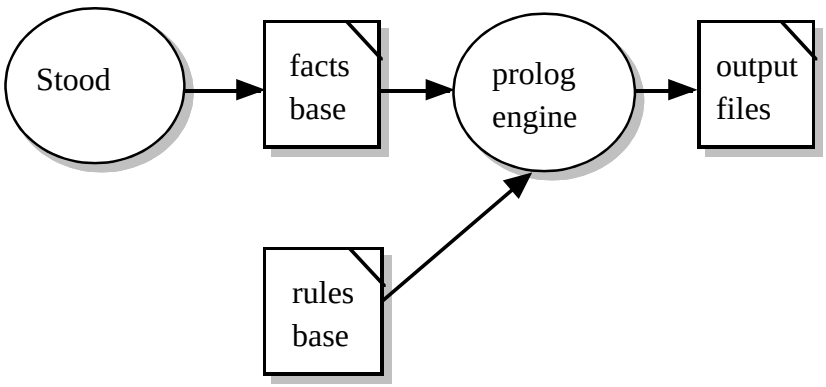
Launching sbprolog executable file:

```
"$STOODBIN/sbprolog"
    -m $SBPROLOG_M_SIZE
    -p $SBPROLOG_P_SIZE
    "$1/_Input.sbp"
```

`STOODBIN`, `STOODPRO`, `SBPROLOG_M_SIZE` and `SBPROLOG_P_SIZE` environment variables are used to provide the actual location of the `bin.xxx` and the `sbprolog` directories, and to set the memory allocation quota for the prolog engine. These variables are set in the initialization file (refer to §2.6).

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 the relevant output directory within current **Application** storage area, before launching the **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

Note that for the reverse engineering operations, the facts base is produced by the source code syntactic analyser in the source code directory.

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

- `isSystem(System).`

`System`	name of the current System

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

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

- `isCurrentRoot(Root).`

`Root`	name of the current Root Component

- `isMissing(Root).`

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

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

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

- `objectLevel(Component,Level).`

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

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

`Feature`	name of a Provided Feature in specified Componen
`Kind`	OPERATION, TYPE, CONSTANT, EXCEPTION
`Component`	name of the Component

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

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

- `isImplementedBy(Pf,Kind,Pc,Cf,Cc,Style).`

`Pf`	name of a Provided Feature of Component Pc
`Kind`	OPERATION, TYPE, CONSTANT, EXCEPTION
`Pc`	name of a Non Terminal Component
`Cf`	name of a Provided Feature of Component Cc
`Cc`	name of a Child Component of Pc
`Style`	1

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

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

- `argument(Op,'OPERATION',Cop,Mode,P,Cty,T,V,K).`

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

- `return(Op,'OPERATION',Cop,Cty,T,K).`

`Op`	name of an Operation of Component Cop
`Cop`	name of a Component in current hierarchy
`Cty`	name of another Component
`T`	name of a Type of Component Cty
`K`	BY_VALUE; BY_POINTER; BY_REFERENCE

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

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

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

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

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

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

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

`Component`	name of an Instance Of Component
`Instance`	actual name of the instance (unused)
`Generic`	name of regarding Generic Component

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

`Feature`	name of a Formal Parameter of specified Generic
`Kind`	OPERATION, TYPE, CONSTANT
`Generic`	name of a Generic of current System

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

`Feature`	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(Component,State,Kind).`

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

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

`Component`	name of a Component in current hierarchy
`Transition`	name of a Transition of specified Component
`From`	name of origin State of specified Transition
`To`	name of destination State of specified Transition
`Event`	name of a Provided Operation of specified Comp.

- `isClass(Type,Component).`

`Type`	name of a Class of specified Component
`Component`	name of a Component in current hierarchy

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

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

- `isInherited(Operation,Component).`

`Operation`	name of an Operation of specified Component
`Component`	name of a Component in current hierarchy

- `inherits(Class,Cc,Superclass,Csc).`

`Class`	name of the Class of Component Cc
`Cc`	name of a Component in current hierarchy
`Superclass`	name of the Class of Component Csc
`Csc`	name of another Component

- `attributes(Type,Ct,Attribute,Ta,Cta,Value).`

`Type`	name of a Type of Component Ct
`Ct`	name of a Component in current hierarchy
`Attribute`	name of an Attribute of specified Type
`Ta`	name of a Type of Component Cta
`Cta`	name of another Component
`Value`	default value for specified Attribute

- `enumeration(Type,Component,Element,Value).`

`Type`	name of a Type of specified Component
`Component`	name of a Component in current hierarchy
`Enumeration`	name of an enumeration element of specified Type
`Value`	default value for specified enumeration element

- `requires(Ccp,Ck,Cmod,Ssymb,Sk,Smod,Ln).`
- `specialrequires(Ccp,Ck,Cmod,Ssymb,Sk,Smod,Ln).`
- `unknownrequires(Ccp,Ck,Cmod,Ssymb,Sk,Smod,Ln).`
- `isRead(Ccp,Ck,Cmod,Ssymb,Sk,Smod,Ln).`
- `isWritten(Ccp,Ck,Cmod,Ssymb,Sk,Smod,Ln).`
- `selfrequires(Ccp,Ck,Cmod,Ln).`

`Ccp`	name of the user Feature
`Ck`	kind of Feature
`Cmod`	name of the user Component
`Ssymb`	name of the required Feature or special symbol
`Sk`	kind of required Feature or special symbol
`Smod`	name of the required Component
`Ln`	logical name of an DataBase section

- `requires(Lang,Ccp,Ck,Cmod,Ssymb,Sk,Smod,Ln).`
- `specialrequires(Lang,Ccp,Ck,Cmod,Ssymb,Sk,Smod,Ln).`
- `unknownrequires(Lang,Ccp,Ck,Cmod,Ssymb,Sk,Smod,Ln).`
- `isRead(Lang,Ccp,Ck,Cmod,Ssymb,Sk,Smod,Ln).`
- `isWritten(Lang,Ccp,Ck,Cmod,Ssymb,Sk,Smod,Ln).`
- `selfrequires(Lang,Ccp,Ck,Cmod,Ln).`

`Lang`	name of a target language
`Ccp`	name of the user Feature
`Ck`	kind of Feature
`Cmod`	name of the user Component
`Ssymb`	name of the required Feature or special symbol
`Sk`	kind of required Feature or special symbol
`Smod`	name of the required Component
`Ln`	logical name of an DataBase section

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

`Component`	name of a Component in current hierarchy
`File`	file pathname
`Ln`	logical name of a DataBase Description section

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

`Feature`	name of a Feature of specified Component
`Kind`	OPERATION, TYPE, CONSTANT, EXCEPTION
`Component`	name of a Component in current hierarchy
`File`	file pathname
`Ln`	logical name of a DataBase Txt section

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

`Feature`	name of a Feature of specified Component
`Kind`	OPERATION, TYPE, CONSTANT, EXCEPTION
`Component`	name of a Component in current hierarchy
`File`	file pathname
`Ln`	logical name of a DataBase default language sectio

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

`Language`	name of a target language
`Feature`	name of a Feature of specified Component
`Kind`	OPERATION, TYPE, CONSTANT, EXCEPTION
`Component`	name of a Component in current hierarchy
`File`	file pathname
`Ln`	logical name of a DataBase 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(Feature,Kind,Component,File,Ln).`

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

- `allocatedRootObject(Design).`

`Design`	name of the logical Root to be deployed

- `allocatedObject(Node,Component).`

`Node`	name of a Virtual Node
`Component`	name of a Component in the logical Root

- `isRequirement(Req,Kind).`

`Req`	name of a Requirement
`Kind`	'?' for unknown, 'D' for derived, 'P' for provided

- `coversRequirement(Req,Ln,Component,Feature).`
- `derivedRequirement(Req,Ln,Component,Feature).`

`Req`	name of a Requirement
`Ln`	logical name of a section
`Component`	name of a Component in current hierarchy
`Feature`	name of a Feature of selected Component

- `hrtPeriod(Component,Mode,F).`
- `hrtOffset(Component,Mode,F).`
- `hrtDeadline(Component,Mode,F).`
- `hrtMinArrivalTime(Component,Mode,F).`
- `hrtPriority(Component,Mode,I).`
- `hrtCeilingPriority(Component,Mode,I).`
- `hrtImportance(Component,Mode,S).`

`Component`	name of a Hard Real-Time Object
`Mode`	'others' or name of a Mode
`I/F/S`	I: Integer, F: Float, S : String

- `hrtWcet(Component,Operation,Mode,F).`

`Component`	name of a Hard Real-Time Object
`Operation`	name of an Operation of selected Component
`Mode`	'others' or name of a Mode
`F`	Float

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(Feature,Kind,Component,Ln,File).`

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

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

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

Note that this predicate is now also produced for the design checkers.

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(Component).`

`Component`	name of a Component for which doc must be create

- `docSection(T,Ln,Pln,L,D,Comp,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
`Comp`	name of a selected Component
`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 Component
`X0`	top left corner abscissa
`Y0`	top left corner ordinate
`X1`	bottom right corner abscissa
`Y1`	bottom right corner ordinate

- `graphicImp(Pc,Pf,Cc,Cf,View,[Xi],[Yi]).`

`Pc`	Parent Component name
`Pf`	Parent Feature name
`Cc`	Child Component name
`Cf`	Child Feature name
`View`	OPERATION, TYPE, CONSTANT,...
`[Xi]`	list of segments abscissa
`[Yi]`	list of segments ordinate

- `graphicUse(Cc,Sc,View,Style,[Xi],[Yi],[Lj]).`

`Cc`	Client Component
`Sc`	Server Component
`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(Component,Label,X0,Y0,X1,Y1).`

`Component`	name of a Component 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(Component,Label,Si,Sd,[Xi],[Yi]).`

`Component`	name of a Component 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

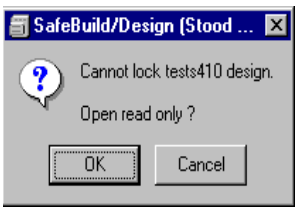
1.5. Unix interface

Stood distribution for **Windows** also contains a `bash` directory, containing a standard **Unix** shell and basic commands implementation for a **PC**. These files come from: `http://sources.redhat.com/`, and are not required if another version of `cygwin` or any other implementation of the required **Unix** commands has already been installed on your platform. Please note that software contained inside the `bash` directory is covered by the **GNU** General Public License (**GPL**). Refer to the **RedHat** web site for further details.

Executable files contained in the `bash` directory should be made accessible by the user's execution path. This can be performed by the appropriate action in the local **Windows** environment, or by extending the current execution path in the `stood.ini` initialization file (refer to §2.6):

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

If this path is not properly set, the following alert box 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	dirname.exe	mv.exe
bash.exe	echo.exe	pwd.exe
cat.exe	gunzip.exe	rm.exe
chmod.exe	gzip.exe	rmdir.exe
cp.exe	hostname.exe	sed.exe
cygwin.dll	ln.exe	tar.exe
date.exe	ls.exe	uname.exe
diff.exe	mkdir.exe	

More recent versions of `cygwin` may be available. The tool administrator may update it directly from **RedHat**, if required. In this case, for compatibility reasons, it may be necessary to recompile the `sbprolog` executable file with the new provided version of the `gcc` compiler, or with another compiler.

2. User’s customizations

Stood may be customized in many ways. Here are described only the easy-to-change options or parameters at user’s level. They are localized in the `.stoodrc` (**Unix**) or `stood.ini` (**Windows**) initialization file. Both files retain the same information, but use a different syntax.

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 the `.stoodrc` file (**Unix**):

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

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

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

These properties may also be set dynamically on the command line.
In this case, the syntax to use is as follow:

```
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` in the working directory
- intermediate: `stood.ini` in the `Windows` or `Winnt` directory or `.stoodrc` in `$HOME`.
- lowest priority (default values): `stood.ini` or `.stoodrc` in the `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:

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

`$TOOL` is the location of the current installation of **Stood**.

`$WORKDIR` is the location from where **Stood** has been 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.

A few specific properties are not described in this section. Their use is mentioned in the appropriate chapters (refer to §1.2.7 for the **http** server settings, §1.2.9 for customized pseudo variables and §3.2.4 for the **Unix** named pipe configuration).

2.2. Changing Applications search path

Stood Applications may be stored into several different directories and may be visible from several simultaneous sessions. They can now be loaded by a direct selection of a `Stood.sto` file with a standard file navigation dialog box. It is also the way an **Application** can now be attached to a **System**. The **System** description files (`.syc`) contain the list of **Applications** that are attached to the **System**.

For portability reasons, it is sometimes more interesting to store in the `.syc` files, the **Application** simple names (instead of their full pathnames). In that case, the way **Stood** knows where to find them is by reading the contents of the `SavePath` property in the `Files` category in the `stood.ini` or `.stoodrc` file. This variable should contain a list of valid pathnames for the current file system, with a few syntactic constraints.

It should be noted that, even on **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 the `SavePath` property. First path of the list will be used as a default directory when creating new **Systems**. 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 rights will be available when creating new **Projects** and **Applications**.

Example:

In the `stood.ini` file, a 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 the current working directory as default saving area for new **Systems.**
- 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.

Note that unlike the previous versions of **Stood**, it is now possible to work on a **Design** which location is not listed into the `SavePath`. However, the `SavePath` is still mandatory to create new **Systems** and to solve ambiguous pathnames in `.syc` files, especially while sharing a **System** along an heterogeneous **Unix**/**Windows** network. For instance, refering to the example above, the file `hoodlib.syc` will be properly loaded from both platforms, even if it contents doesn't specify the full pathnames:

```
SYSTEM_CONFIGURATION IS
    ROOT_OBJECTS
         --|hood/lib|--
END
```

2.3. Customizing target languages

2.3.1. Specifying the default language

Stood is a multi-languages environment. Several implementation languages may be used at the same time for a same **Project**. That's why standard configuration provide access to **Ada**, **C**, **C++** and **AADL** features at the same time for any **Application**. A **pseudo-code** is also available to perform some specific operations. However, 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. Finally, the `MandatoryLanguages` property may used to enforce the use of other languages that the main one.

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

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

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

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

Note that default language may also be changed during an active session by using the *Change design language* command of the *Design* menu. This new default language will be stored with the other **Application** data.

It is also possible to temporarily change the default language when performing language dependent actions (typically: updating a *cross-references table* or checking *design rules*). These local changes are not stored with the other **Application** data.

Note that when using the *Update symbol tables* command of the *Tools* menu, the symbol tables for all installed languages will be updated (not only for the default language).

2.3.2. Interfacing with compilers

Stood offers advanced features to perform source code generation and reverse engineering. These operations require some knowledge about the various compiling environments that are available on the platform. They may have to be customized by the tool administrator.

A few environment variables are used by some internal tools to call a compiler after a source code generation has been completed. The `ADA_PATH`, `C_PATH` and `CPP_PATH` properties in the `Environment` category may be used to specify the location of the compilers to be called. This customization is not required if the relevant pathnames have already been included in the default execution path of the system.

Another environment variable is proposed to customize the command line of the **C** reverse engineering pre processing. It is generally necessary to include a few additional options for thr **C** pre processor to find the appropriate header files that are included to the source files that are to be reversed. The `REVERSE_OPTIONS` property in the `Environment` category must be used for this purpose:

```
Environment.REVERSE_OPTIONS:-I/usr/X11R6/include
```

The `Languages` category may be used to customize the source file suffix for the various languages that are supported by **Stood**. The default values are:

PSEUDOSPECSUFFIX	.s
PSEUDOBODYSUFFIX	.b
CSPECSUFFIX	.h
CBODYSUFFIX	.c
CPPSPECSUFFIX	.h
CPPBODYSUFFIX	.cc
ADASPECSUFFIX	.ads
ADABODYSUFFIX	.adb
AADLSUFFIX	.aadl

Note that the name of these properties for each language may be changed in the `extractors` file of the `config/code_extractors` configuration subdirectory, and their value are also used by the language suffix pseudo variables in the DataBase file (refer to §1.2.9).

When **Stood** generates source code, the previously generated files are cleaned up from the target directory. However, in the case of a partial generation of the code, it is necessary to specify which files mustn't be removed (as they won't be generated again). The `NoCleanUpFor` property in the `Languages` category is used to specify the name of a code generation **pragma**. If this **pragma** is allocated to a **Component** (or a set of **Components**) during the code generation process, then the corresponding source files won't be cleaned up. By default, the **pragma** `except` is used to identify the **Components** that musn't be generated, and thus which files musn't be cleaned up.

2.4. Customizing the main window

Unlike the previous versions of the tool **Stood** 5.0 concentrate most of its features in a unique main window. However, this window shows various views that are controlled by a set of tabs. Each view provides a different button bar. It is possible to customize the name and the button bar of each view by editing the `stood.ini` or `.stoodrc` file. In addition, default location and size of the main window on the screen may be predefined:

For this purpose, each view must be referenced by its predefined identifier:

`req`	requirements editor view
`gra_hood`	HOOD graphical editor view
`gra_uml`	UML graphical editor view
`txt`	textual editor view
`hie`	inheritance tree
`chk`	design verification view
`ext`	code generator view
`code`	code editor view
`rev`	code reversor view
`doc`	documentation generator view
`vna`	design allocation editor view

Note that it is no more possible to customize the button bar for the **S**tate **T**ransition **D**iagram editor (`std`), the inheritance tree (`hie`) and the call and access trees (`utr`), and the following windows don't exist any more due to the new layout, it is thus no more needed to define a button bar for the previous main window (`main`), the previous system editor (`syc`), the previous cross references window (`crf`), which has been inserted into the design verification view and the previous documentation schemes window (`sch`), which has been inserted into the documentation generator view.

2.4.1. Customizing view names

The `Views` category can be used to specify a name to each tab of the main window. This may be useful to better fit alternate software development standard terminology. The property names are the view identifiers that are specified in the previous paragraph, except for the **HOOD** and **UML** graphical editors that are controlled by a unique tab that must be referenced with the identifier `gra`.

Note that any renaming of these tabs also impacts the corresponding *Tools* menu items. Take care to consider these changes in **STShell** scripts respect, as they may use these tab names and menu items in their command parameters.

The defaults for tabs are the following:
in the `stood.ini` file:

```
[Views]
req=Requirements
gra=Graphic Design
txt=Detailed Design
chk=Checkers
ext=Code
doc=Documentation
vna=Deployment
```

in the `.stoodrc` file:

```
Views.req:Requirements
Views.gra:Graphic Design
Views.txt:Detailed Design
...
```

2.4.2. Customizing buttons

The `Buttons` category can be used to customize the button bar for each view. The property name is the identifier of the view as they are defined in the table above. The syntax to be used to specify a button bar is similar to the one used in the previous versions of **Stood**. However, the previous definitions are no more relevant as the references to menus are completely different. Another difference is that a procedure number may be given instead of a menu reference. This is especially necessary to differenciate contextual create actions from the generic ones that are provided in the menu. With the former ones, user interaction is required, not with the latter ones. The list of valid procedure numbers for button bar definition is provided below.

Syntax of a buttons bar definition in a `stood.ini` file:

```
[Buttons]
V1=L11,M11,I11,...; ... ;L1n,M1n,I1n,...
...
Vm=Lm1,Mm1,Im1,...; ... ;Lmn,Mmn,Imn,...
```

Syntax of a buttons bar definition in a `.stoodrc` file:

```
Buttons.V1:L11,M11,I11,...; ... ;L1n,M1n,I1n,...
...
Buttons.Vm:Lm1,Mm1,Im1,...; ... ;Lmn,Mmn,Imn,...
```

Where:

`Lij`	label to be displayed in the button bar baloon
`Mij`	position of the menu in the menu bar or procedure number
`Iij,...`	position of the item in the menu, followed by submenus if any

Notes:

- Additional semi-colons may be used to increase separation space.
- When a label begins with a `*`, then the icon of the same name (refer to §1.2.6) will be displayed instead of the label name.

The list of procedures that may be used in buttons bar definitions is the following:

`proc501`	new HOOD object
`proc502`	new HOOD class
`proc503`	new HOOD cyclic
`proc504`	new HOOD sporadic
`proc505`	new HOOD protected
`proc520`	new HOOD feature
`proc530`	new HOOD set
`proc540`	new HOOD use connection
`proc541`	new HOOD implementedBy connection
`proc542`	new HOOD inheritance
`proc543`	new HOOD aggregation
`proc601`	new UML component
`proc602`	new UML class
`proc603`	new UML cyclic component
`proc604`	new UML sporadic component
`proc605`	new UML protected component
`proc620`	new UML feature
`proc640`	new UML assembly
`proc641`	new UML delegate provided
`proc642`	new UML inheritance
`proc643`	new UML aggregation
`proc644`	new UML delegate required

2.4.3. Customizing default position, size and zoom

The default position and size of the main window as well as zooming parameters and default size of graphical boxes may be specified in the `Window` category. The definition of the `Position`, `Extent` and `NewBoxExtent` properties must comply with the 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.
- `NewBoxExtent` property specifies the default size of newly created boxes.
- negative values are allowed.

The zooming options of the graphical views may be customized by the `InitialZoom` and `ZoomIncrement` properties in the `Window` category. Their value must be given in percentage. It may be useful to change these values due to best fit the resolution of the screen.

2.5. Changing default fonts and colors

It is possible to configure a few fonts and colors that are directly controled by **Stood**. This configuration will be performed by setting a few properties inside the `stood.ini` or `.stoodrc` file. These properties belong to the `Fonts` and `Colors` categories respectively. On **Unix** platforms, the **Motif** widgets that are used by **Stood** can also be customized. The corresponding resources must simply be overloaded in the `.stoodrc` file.

2.5.1. Customizing fonts

Properties name for fonts customization are:

`DefaultFont`	font to be used by default.
`DiagramFont`	font to be used in graphical diagrams.
`TreeFont`	font to be used used in graphical trees.
`TEXT`	font to be used in informal sections.
`CODE`	font to be used in code sections.

The value for `font` properties must be a valid font name and size that is available on the current platform. All the other fonts (menus, lists, ...) are controled by the window manager, and should be customized by any appropriate procedures in **Windows** control panel or **Motif** ressource files. On **Unix** workstations, a `Stood` **Motif** resources file for **Stood** may be optionally created in any of these locations (none is provided with the standard distribution):

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

It is also possible to introduce **Motif** resources directly inside the `.stoodrc` initialization file to control the widgets appearance, as shown in the example below.

Example:

A possible `stood.ini` font configuration is:

```
[Fonts]
DefaultFont=Arial 9
DiagramFont=Comic Sans MS 10
TreeFont=Comic Sans MS 10
TEXT=Times New Roman 12
CODE=Courier New 12
```

A possible `.stoodrc` font configuration is:

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

*fontList: -adobe-helvetica-medium-r-normal
--10-100-75-75-p-56-iso8859-1
*XmText*fontList: -adobe-helvetica-medium-r-normal
--10-100-75-75-p-56-iso8859-1
*XmTextField*fontList: -adobe-helvetica-medium-r-
normal--10-100-75-75-p-56-iso8859-1
```

2.5.2. Customizing colors

Property names for color customization are:

`Module`	Component box in the HOOD diagrams
`ModuleExport`	exported Component box in the HOOD diagr
`Component`	Component name in the HOOD diagrams
`ConnectionUse`	Use relationship in the HOOD diagrams
`ConnectionImpl`	Implemented_By link in the HOOD diagrams
`ConnectionLabel`	DataFlow, Exception Flows labels
`State`	State box in the State-Transition Diagrams
`Transition`	Transition in the State-Transition Diagrams
`TransitionLabel`	labels on Transitions in the S-T Diagrams

The 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. It is also possible to customize **Motif** resources inside the `.stoodrc` initialization file, or inside a dedicated file, to control the widgets appearance, as shown in the example below.

Example:

A typical `stood.ini` color configuration is:

```
[Colors]
Module=0 0 128
ModuleExport=192 192 192
Component=0 0 255
ConnectionUse=9 117 18
ConnectionImpl=255 153 0
ConnectionLabel=0 0 128
State=0 0 128
Transition=255 0 0
TransitionLabel=0 0 255
```

The corresponding `.stoodrc` color configuration is:

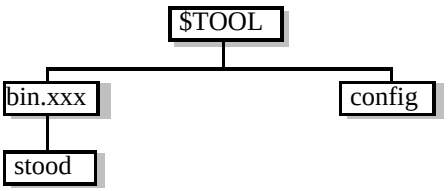
```
Colors.Module:0 0 128
Colors.ModuleExport:192 192 192
Colors.Component:0 0 255
Colors.ConnectionUse:9 117 18
Colors.ConnectionImpl:255 153 0
Colors.ConnectionLabel:0 0 128
Colors.State:0 0 128
Colors.Transition:255 0 0
Colors.TransitionLabel:0 0 255

*OverrideShell*background: LightYellow
*XmText*background: White
*XmTextField*background: White
*XmList*background: White
```

2.6. Customizing the environment

A few properties may be changed to customize the 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 the `Files` category can be modified to let **Stood** point to another configuration directory. Default value is `$TOOL/config`, that is the `config` directory located in the same parent directory as the current `bin.xxx` directory.



When using its internal or external tools, **Stood** needs to launch **Unix** shell scripts (even under a **Windows** environment). The `Shell` property in the `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.

A few **Unix** environment variables are required by **Stood** post-processors (rules checkers, code extractors, documentation generators). These variables may be directly set within the `Environment` category. Defaults values are `$TOOL/sbprolog` for the `STOODPRO` variable, and `$TOOL/bin.xxx` for the `STOODBIN` variable.

Additionally, the SBPROLOG_M_SIZE and SBPROLOG_P_SIZE variables may be set to specify the memory allocation requirements (in bytes) for the prolog engine that is used by the post-processors.

Note that other **Unix** or **Windows** environment variables may bet set if required. For instance, it may be necessary to extend the execution path to give access to specific executable files:

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

Licensing information is also specified by several properties belonging to Licensing, License or FlexLM categories. Please refer to *Installation Manual* or contact your system administrator or **TNI Europe**'s technical support if you need to set or change these properties. Please note that these three categories are exclusive, and mustn't be set together:

```
[Licensing]
Organization=Evaluation
Licensee=None
LicenseCount=1
Mode=Full
ExpirationDate=31/12/2004
Password=6227029

[NFL]
File=\\hostname\tools\license\stood.nfl
ReleaseDelay=1440

[FlexLM]
File=\\hostname\tools\license\license.dat
```

2.7. Other simple customizations

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

- the `Welcome` property in the `General` category specifies the string to be displayed on top of *main editor*. The default value is `Stood 5.0`. It is an easy way to identify a particular configuration.

- the `ShowDirectories` property in the `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.

- the `GraphRepresentation` property in the `General` category specifies whether the trees must be displayed as a textual lists, or a graphical trees. Values are `List` or `Tree`.

- the `GraphSizeLimit` property in the `General` category specifies the maximum size of a tree to be printed graphically in the design documentation. For readability reasons, all the trees which size exceed this limit will be inserted textually in the documentation.

- the `DefaultGraphics` property in the `General` category specifies the default graphical notion for the architectural diagrams. Values are `HOOD` or `UML`.

- the `Default` property in the `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`, `mif`, `html`, `pdf`.

- the `UniqueNameSpace` property in the `General` category specifies if multiple namespaces are allowed in the code or not. By default, each **HOOD Component** defines a separate namespace.

- the `EnableSTShellMenu` property in the `Security` category may be used to invalidate the *STShell* item in the *Windows* menu of the *main editor* to forbid the execution of **STShell** commands for security reasons. Default value is `Yes`.

- the `MarkExportedModule` property in the `General` category defines whether a **Component** keeps its exported attribute or not in the original **Design**. When set to `Yes`, the corresponding box has greyed borders when the **Component** has been exported. Default value is `Yes`.

- the `CompleteCrossReferences` property in the `General` category controls the way symbol tables are stored and call trees are drawn. When set to `No`, symbol tables and call trees are similar to those of the previous versions of **Stood**. When set to `Yes`, all the occurences of calls are stored and call trees show accessed **Data** via **Operation** parameters. Default is `Yes`.

- the `ReplaceDashBy` property in the `General` category gives the replacement characters for dash characters found inside a **SIF** filename when associating it to a **Root Component** name. This is especially useful while reversing **Ada** code containing child units. Default is the underscore character.

- the `EnableFirstUnderscore` property in the `General` category allows symbols with an underscore as their first character to be recognized in the symbol tables. This is allowed in **C** but not in **Ada**. Note that the corresponding lexical analysers must follow the same rule. Default is `No`.

- the `KeepPseudoPrefix` property in the `General` category activates the processing of the dot notation in **Pseudo** code sections (like in **Ada** code sections). Default is `No`.

- the `DirectoryEdit` property of the `General` category is used to specify which application must be launched to display the contents of the directories from the *Tools* menu. Default is `explorer.exe` on **Windows** and an appropriate `xterm` command on **Unix**.

- the `StatusTimeOut` property of the `General` category specifies the duration of the red display of a new error in the *status bar*.

2.8. Configuration management

Stood database can't manage directly different configurations or versions of a same **Application**, but may interact with external configuration and version management systems. Two kinds of interfaces are proposed: identification tags and checkin/checkout procedures

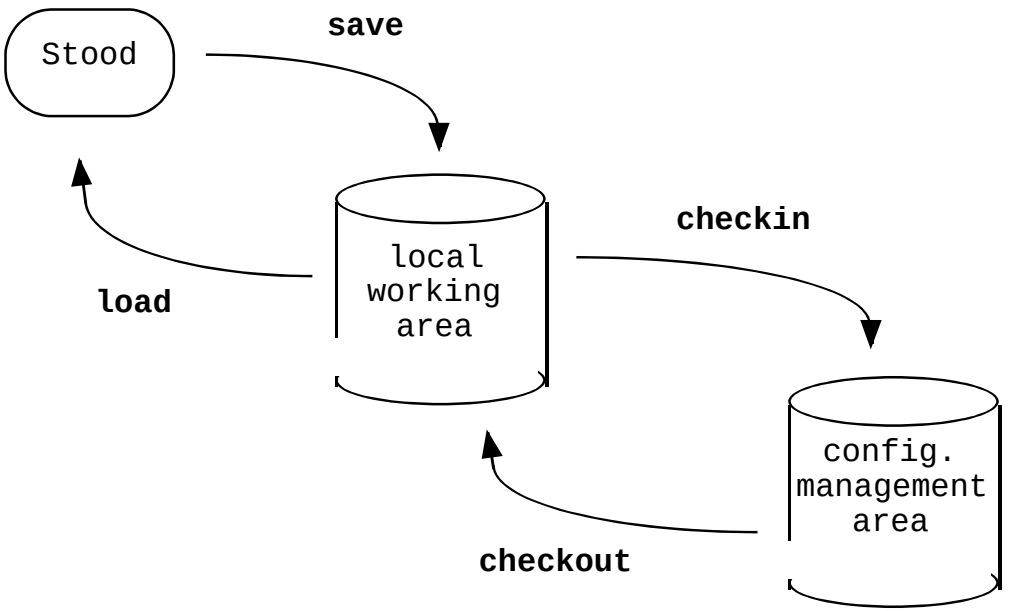
2.8.1. Identification tags

It is possible to ask **Stood** to automatically insert an identification tag in all the files that are stored in an **Application** database for configuration management purpose. The tag must be added manually for the files that are edited by hand. In the other items, the tag will be included automatically, and between appropriate comment separators if necessary. A generic tag value may be specified in the `Header` property of the `Versioning` category. Default values are blank to specify that no tag has to be inserted, or `$Header$` else. This tag may be processed by configuration management systems like **RCS**.

2.8.2. Checkin/checkout procedures

Shell scripts may be customized to interface with configuration management systems like **CVS**. These scripts are stored in the `config/internalTools` configuration subdirectory. As several concurrent interfaces may be present, the actual scripts to use for a given session are specified by the following properties of the `ConfigurationManagement` category:

`CheckInProcedure`	from local space to conf. management area
`CheckOutProcedure`	from conf. management area to local area
`CheckLockProcedure`	lock in the conf. management area
`CheckUnlockProcedure`	unlock in the conf. management area



When a **Design**, a **Component** or a **Property** is loaded, relevant files contents are copied from the local working disk area to the **Stood** memory. When a **Design**, a **Component** or a **Property** is saved, data is stored into the corresponding files in the local working disk area. If a **Design** or a **Component** is loaded in read-write mode, then a lock file (`Stood.lok`) is created in the local working area

During a session, the local working disk area and the **Stood** memory will be updated by the contents of the configuration management area, when a **Design** or **Component** *checkout* menu command is used. Similarly, the configuration management area, and the local working area, will be updated by the current contents of the **Stood** memory, when the **Design** or **Component** *checkin* menu command is used. If the *lock* checkbox of the *checkout* or *checkin* dialog is set, then the `checklock` or `checkunlock` scripts will be also activated.

The `ConfMgrActivated` property in the `File` category may be used to activate or deactivate the use of the configuration management scripts at a **Design** level. When the scripts are activated, the name of the configuration management system to which **Stood** is interfaced, may be specified in the `ConfMgrLabel` property of the `File` category.

For the configuration management interface to also work at a **Component** level, the property `ConfMgrModular` in the `File` category, must also be set to `Yes`. In that case, due to the hierarchical structure of the model, a single *checkout* or *checkin* command may generate a sequence of call of the respective scripts, one for each **Subcomponent**. An option consists in calling the script only once, and send the list of concerned **Subcomponents** in a file, which name is stored into the last parameter passed to the script. To activate this last option, the `CheckOutWithArgFile` and `CheckInWithArgFile` in the `ConfigurationManagement` category must be set to `Yes`. Default is `No`.

Version labels may be defined in the `Versions` property of the `ConfigurationManagement` category. The label, that must be selected in the appropriate dialog box, may be used to select or save a given version.

When **CVS** is used, the configuration management information is stored in a `CVS` subdirectory located in each directory of the **Application** database. To prevent **Stood** to alter this information, the `CheckKeepingFiles` property of the `ConfigurationManagement` category must be used.

Note: The configuration management interface at a **Component** level is a new feature of **Stood 5.0**. When using older configuration files, the required `COMPONENTTRASHDIRECTORY` section in the `DataBase` descriptor file may be missing. In that case, please contact the technical support.

2.9. Requirements management

Stood may load a list of requirements from **REQTIFY**, the requirements traceability tool of the **SafeBuild** suite. In addition, the lexical definition of a requirement or a reference to a requirement may be customized for the current **Project**.

The `REQTIFY_PATH` property in the `Environment` category must be used to set an environment variable to be used in the `reqtify.sh` shell script located in the `config/internalTools` configuration subdirectory. This script calls **REQTIFY** to get the proper list of requirements. The path must specify the actual location of the main executable file for **REQTIFY**.

Most requirements will be loaded from the previous requirements analysis tasks. Definition of a new requirement during the design process, will be a derived requirement, and its syntax must be specified with the `Define` property of the `Requirements` category.

Design entities must cover requirements. To define the syntax of a reference to a requirement, the `Reference` property of the `Requirements` category must be used. Both `Define` and `Reference` properties must be specified with standard regular expressions. Note that the `\` escape character must be dupplicated in the `.stoodrc` file.

The `IgnoreCase` property of the `Requirements` category specifies if the recognition of requirements must be case sensitive or not, and the `CatalogSuffix` property of the `Requirements` category gives the default suffix for the file selector when importing requirements. Possible values are `.txt` for plain text files or `.rqtf` for **REQTIFY** projects.

Example:

If the requirements analysis process of the project defines a list of requirements of the following form:

```
REQ_001
REQ_002
REQ_003
...
```

And the coverage of a requirement in the design is defined by a `cf.REQ_xxx` expression, and the definition of a derived requirement by a `def.REQ_yyy` expression, then the `Define` and `Reference` properties could be set as follow:

in the `stood.ini` file:

```
[Requirements]
Define=\(def\.(REQ_[^\)]+)\)
Reference=\(cf\.(REQ_[^\)]+)\)
```

in the `.stoodrc file`:

```
Requirements.Define:\\(def\\.(REQ_[^\\)]+)\\)
Requirements.Reference:\\(cf\\.(REQ_[^\\)]+)\\)
```

3. Launching Stood

Stood may be started in four different modes:

- interactive mode (usual mode)
- semi-interactive mode
- batch mode
- remote control mode

The interactive mode is the only one which requires interactive use of a terminal keyboard and mouse. With the three other modes, **Stood** can be controled by a sequential list of instructions. These instructions must be written with a specific syntax, in a language called **STShell**, and define an Application Programming Interface (**API**) for **Stood**.

When **Stood** is started, an instance of the *main window* is shown on the screen. It gives access to the full range of menus, selections and graphical actions that is necessary for an interactive usage of the tool. Please refer to the appropriate contextual help or user's documentation to get information about the use of **Stood** in interactive mode.

3.1. STShell

A **STShell** instruction is a command to be executed by **Stood**, and generally includes a list of parameters. Its 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.

3.1.1. STShell parameters

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

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

The use of the `*` 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 or view 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.

`main`	main window
`tre`	design tree
`hie`	inheritance tree
`gra_hood`	HOOD graphical editor
`gra_uml`	UML graphical editor
`std`	states-transitions diagram editor
`txt`	text editor
`crf`	cross-references table
`utr`	call tree
`sch`	documentation scheme editor
`req`	requirements 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
`dbconf`	configuration management dialog box
`last`	last opened window

The parameters may also need to reference a list in a browser. Each list is identified by an integer.

Notes:
- Only one window of each kind may be referenced at a time within a sequence of **STShell** instructions.
- All the parameters referencing a menu, a menu item, a list, a list element and a button name should match exactly the name shown by the **Stood** windows. However, space characters at the beginning or the end and suspension points in menus may be avoided

3.1.2. STShell instructions

The following instructions 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 command to perform a predefined list of lower level actions.

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

- `Context("project","application"[,"component"])` : select the given **Project** and **Application**, and optionally the given **Component** in the top left selection list of the *main window*.

- `Feature("feature"[,"property"])` : select the given **Feature**, and optionally the given **Property** in the bottom left selection list of the *main window*. A **Feature** must be an **Operation**, **Type**, **Constant**, **Data**, **State**, **Transition** or a *checker* rule category

- `Property("property"[,"feature"])` : select the given **Property**, and optionally the given **Feature** in the bottom left selection list of the *main window*. A **Property** must be referenced by its *logical name*, as defined in the *DataBase* descriptor file (refer to §1.2.9)

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

`id`	window identifier (usually: main)
`menu`	Menu name in window menu bar
`item`	Item name or position integer index in menu
`subitems`	Items in submenus, if any

- `Menu("id","menu","item","box",X1,Y1,X2,Y2)` : create a new box at the specified coordinates in the given graphical editor.

`id`	window identifier (gra_hood, gra_uml or std)
`menu`	Menu name in window menu bar
`item`	box creation Item name in menu
`box`	name of the box to be created
`X1`	left coordinate
`Y1`	top coordinate
`X2`	right coordinate
`Y2`	bottom coordinate

- `Menu("id","menu","item"[,"subitems"],"C"[,"F"])` : create a new connection, to specified destination.

`id`	window identifier (usually: main)
`menu`	Menu name in window menu bar
`item`	connexion creation Item name in menu
`subitems`	connexion creation items in submenus, if any
`C`	destination Component or State for the connexion
`F`	destination Feature in destination Component, if an

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

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

- `ListMenu("id",list,"item"[,"subitems"])` : execute a given item, or its subitem if any, 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
`subitems`	items in submenus, if any

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

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

`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.

- `Delay(duration)` : wait for the specified number of seconds.

- `Show(x,y,"text",duration)` : display the given text at the given coordinates during the given number of seconds. The origin is the top left corner of the screen.

- `BoxSelect("id","box"[,"feature"])` : select the specified box (**Component** or **State**) or its specified **Feature** if any, in the given graphical editor.

`id`	window identifier (gra_hood, gra_uml or std)
`box`	Component or State name
`feature`	optional Operation, Type, etc...

- `Write("id","text")` : write the specified text in the currently selected text input area of the current view. Use the `Context` and `Property` instructions first, to select the right text input area.

`id`	window identifier (txt)
`text`	text to be inserted

- `Use("id","origin","dest","dir1","dir2")` : draw a **Use** relationship between the two specified **Components**.

`id`	window identifier (gra_hood or gra_uml)
`origin`	origin Component name
`dest`	destination Component name
`dir1`	direction at origin: N,E,S or W
`dir2`	direction at destination: N,E,S or W

- `ImplementedBy("id","origin","dest","child")` : draw an **Implemented_By** relationship between the specified **Features** from the current parent **Component** to the specified child **Component**.

`id`	window identifier (gra_hood or gra_uml)
`origin`	origin Feature name
`dest`	destination Feature name
`child`	child Component name

Note: The **STShell** commands `Button` and `TabSelect` are no more supported. This is due to the new layout of the main windows and to the reorganization of the main menu bar.

3.1.3. STShell program example

```
#--------------------------------------
# BATCH CODE GENERATION EXAMPLE
# stood v5.0 - TNI Europe - August 2004
#--------------------------------------
#
# Select "test" design inside "tests" system :
Context(tests,test)
#
# Select the "code" view in the "main" window :
Menu(main,tools,view,code)
#
# Launch the code generation :
Menu(main,tools,code,"full extraction")
Click(dbobj,OK)
#
# Show "extraction messages" :
Property("ada::ExtractMessages")
#
# Open a remote editor on that file :
Menu(main,tools,"external tools",emacs)
#
# Quit stood :
Menu(main,file,quit)
#--------------------------------------
```

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

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)

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
```

To open Stood on an existing System (file with .syc extension), enter:

```
stood -s filename.syc
```

To open Stood on an existing Root (file with .sto extension), enter:

```
stood -r filename.sto
```

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
```

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 the commands sequence. To launch **Stood** in batch mode, enter:

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

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

3.2.4. Remote control mode

On **Unix** platforms, it is possible to send **STShell** commands to an active session of **Stood**. An input named 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,tools,view,"graphic design")' > st
echo 'BoxSelect(gra_uml,box)' > 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.
- To enable the named pipe, the following properties must be present in the `.stoodrc` initialization file:

```
Server.Name:st
Server.DisableSTShellPipe:No
```

Additionally, a **DDE** port is also initialized by **Stood** for remote control. This mode is mainly used on **Windows** platforms, but may also be operated with Unix environments. **STShell** instructions can be sent to the **DDE** port of **Stood**.

Finally, **Stood** can operate as an **http** server. Please refer to §1.2.7 for further details.

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.

www.tni-world.com
stood@tni-world.com

TNI Europe
Triad House
Mountbatten Court
Worall Street
Congleton
Cheshire
CW12 1AG
UK

+44 1260 291 449

Ellidiss Technologies
Technopôle Brest-Iroise
115 rue Claude Chappe
29280 Plouzané
Brittany
France

+33 298 451 870