SUPERVISOR program User Guide
SV version 0.24

Kornilov V., Shatsky N., Voziakova O.

March 22, 2004

Contents

List of Figures

Introduction

In this document we consider the use of Supervisor program for automation of the observations sequence with a number of component programs which are running the observational equipment (telescope, detectors etc). The basic principles and the system tuning and running is described below, whereas the MASS/DIMM scenario description, communication protocol accepted in MASS/DIMM and programming topics are located in appendices. Below we briefly repeat the general ideas of the SV-driven system explained in also in the Report I. on the "Combined MASS/DIMM instrument" document (Kornilov et al, 2003).

1 Basic principles

The observational system from the software point of view consists of a number of driver-programs running on a single or several machines in a local network. Each driver-program is called "component" and allows to manipulate with a single hardware component of the system – namely telescope, some detector, stand-alone meteo-service or anything else. The component normally has two interfaces: local (console or graphic) user interface which allows to work manually with the device, and the command interface for communication with the program via a socket-connection. The latter is used by the Supervisor program, which runs the specially user-written system-tuned scripts. These scripts consist of a number of commands sent to the components when some particular action or result is needed from them. Components start the requested action or reply immediately with the ready result to the requesting agent (Supervisor). The protocol according to which they should act and accept commands is described in appendix. The main content of latter document specifies the demands to the MASS/DIMM system specially.

Normally, the observational set is a continuous period of time (say, a night, week or a year) during which Supervisor must solve two tasks:

1. Tracing the conditions – whether they are good or not for starting observations;

2. Start observations when conditions are good and stop them (promptly!) when they become bad.

The scientific results collection is not the primary task of Supervisor, it only provides the extensive logging of own actions and commands exchange.

The listed tasks are almost independent from each other, both in a sense of used components and in a timing. They normally are run asynchronously and in parallel.

In other words, SV provides the tools for sequencing the commands to the components in order to provide the non-excessive and efficient monitoring of observational conditions – this is the scenario of the "Circumstance monitor" – and logical and optimal chain of commands to detectors, telescope etc for conducting the observations – this is the "Observations" scenario.

Both scenaria are written as Tcl scripts, but run not in a pure Tcl interpreter, but from the interpreters started by SV. User must provide the names of these script files (say, `circmon.tcl` for a monitor and `obs.tcl` for observations scenario) together with the component parameters in the configuration file of SV – this is the `sv.cfg` file. Latter file may contain some parameters which tune the performance of the scenaria themselves – in order to easily modify their characteristics without edition of scenaria with a dully search for hard-coded numbers. Once provided such a good configuration file, user may start the components, ensure that they are in a good shape, start SV, and ensure that its start-up sequence has passed well and all components were successfully connected to SV. That's all. In ideal situation, the User may come back home and wait for the e-mails from SV sent in case of occurrence of some error situations during observations (sending a mail is a usual option for the `emergency_sys` script in CFG).

2 Component programs

The current release of SV (or, more precisely, the provided scenaria of circumstance monitor and observations) operates with following component programs:

a) OBJM – Object Manager (Tcl script `objm.tcl`). The tool for selection of the best-viewed star according to the user-provided selection criteria and the star catalogue. Includes the functionality of CIRC.

b) CIRC – Circumstance Monitor (Tcl script `circ.tcl`). The computer of astronomical circumstances – the Sun and Moon equatorial coordinates and altitudes above horizon, stellar and Universal time etc. Included into OBJM since it supports all the astronomical calculations needed for objects selection.

c) METEO – Meteo service simulator (Tcl script `meteo.tcl`). To be replaced with a real meteo-gathering program or upgraded to make the real conditions requests. Currently conditions are simulated using the random number generator.

d) TLSP – Telescope driver simulator (Tcl script `tlsp.tcl`). To be replaced with a real driver program, or upgraded to send the commands to the mounting when it is needed. Currently, the telescope activity is simulated by some waiting periods.

e) DIMM – the Differential Image Motion Monitor program simulator (Tcl script `dimm.tcl`). To be replaced by a real DIMM program, e.g `Robodimm` for Windows or whatsoever appears. In case of Robodimm, it will run on a separated Windows machine. Current simulator supports all the needed SV commands to DIMM replacing the real work for measurements with waiting periods of 60 to 120 sec length (random). The simulator also supports the centering device functionality reporting the star shifts in arcseconds on request. Shifts are random numbers for both coordinates which range increases from (-1,1) to (-100,100) arcsec when the time since the last request increases from 1 to 100 seconds and further. The CENT-functions run independently from DIMM functions in current simulator.

f) MASS – the Multi-Aperture Scintillation Sensor program simulator (Tcl script `mass.tcl`). To be replaced with a current release of Turbina program (version 2.04 or later) running on the same machine as SV or on the other Linux machine. Simulator provided serves thus only for purposes of SV testing waiting for 60 sec upon RUN command for main measurements and for 5 sec upon RUN SCEN1 command for background measurement. The requests for data with GET-commands are supported (see MASS User Guide, Version 2.04).

These components (mostly simulators) may be started together using the provided `start_dummies.sh` script and killed by the `stop_dummies.sh` script which is created by `start_dummies.sh`.

When one wants to replace the simulators with real component programs, the `sv.cfg` file must be modified respectively to change the `port`, `host` and `ident` parameters of them. Then, to use other non-replaced components, the line

```
set comps "mass dimm meteo objm tlsp"
```

in `start_dummies.sh` must be edited by removal of the names of replaced components.

3 Writing the scenaria

Circumstance monitor and Observations scenaria are normally relatively simple scripts. They do not pay attention to errors which might occur in components during execution – this is the competence of Supervisor (see also the end of this section). Also they must not care about the connection issues of the components – they only know the new Tcl commands `cmd, is_cmd, wait_cmd, wait_sec, initialize, stop_park` and *comp1, comp2... compn* for communication with them (where *comp1...* are the component program names listed in the configuration – the commands suited to return their configuration and requested parameter values).

Both observations and circumstance monitor scenaria MUST take care about the initialization or parking of the components before and after working with them, respectively. Note that this is a demand of SV versions starting with **0.20**; earlier versions initialized and parked components automatically. For doing so, the commands (actually, macros) `initialize` and `stop_park` are provided – the scenario may use them as well as their own scripting. Initialization should be done before scenario starts execution of measurements – i.e. before sending any commands implying the non-parked status of components.

Parking while stopping the scenario is more complex: the scenario has no idea when it may happen. If some actions must be undertaken while shutting down the observations (parking normally, also some possible after-scripts cancellation etc), the procedure with the reserved name `end` (without parameters) must be written in the scenario. In case the scenario is stopped, SV searches for such a procedure and, if found, executes it before deletion of the scenario interpreter. First thing which this procedure must do is to prevent any other procedure from sending the commands to components (organized, for example, with help of after-constructs), and then to call `stop_park` *"comp.list"* if this is needed.

Summarizing, the scenaria scripts do have beginning, but not finalization which is provided by an isolated procedure `end`. Finally, Observations scenario has no idea about existence of Circumstance monitor scenario, and the latter knows only the commands `start_obs, is_observations_now` and `stop_obs` to manipulate the former.

Below we sketch out the approximate sequence of actions which the scenaria should perform in order to make observations:

Circumstance monitor:

1. initialize monitoring components (e.g. `initialize "CIRC METEO"`)
2. set the needed parameters in them (optional; e.g.: `cmd CIRC set longitude=..`)
3. organize periodical requests of weather and sun-height conditions and, according to results and current state of observations (run or not), start or stop observations.

Observations:

1. ask the new object parameters

2. point the telescope to new object. Check that it is found with help of centering component (coordinate difference is not reserved +999 for the case of NOSTAR-error)

   If found

3. center the object in field of view with help of centering comp.

4. run a measurement command sequence

5. if it is a time to check background or make some calibration – do it with help
   . of telescope and/or detectors

6. trace whether it is enough with current object.
   . If yes – go to p.1,
   . else – continue with p.4

   else

7. try to find an object around the expected position with telescope and centering
   . device.
   . If found – check that it's likely the searched star, go to p.3.
   . else – wait for some time and go to p.1

As we see, observations scenario is much more complicated as it is. Writing the correct scenario is a key-point of successful usage of SV in automated observations.

To write a scenario, the user should have at least general knowledge of the Tcl language, but study of the sample `circmon.tcl` and `obs_massdimm.tcl` scripts may help to do the job without one since Tcl is very simple. Basic points are:

- each line is a command with command name being the first word
- command is ended by ";"-character or newline. The command may be continued on a second line with help of trailing "\"-character.
- a command in brackets `[command args]` represents the command substitution: e.q. `[OBJM port]` results in "1234" if `sv.cfg` has a line "port 1234" in "component OBJM" section. This substitution may stand as an argument to another command: e.g.

  ```
  cmd OBJM set longitude="[CIRC longitude]"
  ```

  Here double-quotes are part of the command, since the result of CIRC-command may contain (and contains!) the spaces.
- braces "{...}" are used to group the arguments. The content of braces may be placed on several lines without trailing backslashes. Most common use of this is conditional commands `if {condition} then {what then} else {what else}`.
- variables may be declared on place of initialization where they become needed. Do not forget that assignment may be done only as:

  ```
  set a 1; set fi [CIRC latitude]
  ```

  and NOT as

  ```
  a = 1; fi=[CIRC latitude] - THIS IS WRONG.
  ```

- comments (#-started lines) are also commands - they are interpreted but only not executed. DO NOT PUT non-paired braces or brackets in comments! Avoid comments in switch-commands!

The list of commands which user may use in scenaria is given in sections "SAFE INTERPRETERS" of the manual page to Tcl `interp` command (see `man n interp`). That's the starting point for a beginner. All commands listed there may be studied using their own manual pages (section "n"): e.g. `man n after`.

The syntax of SV special commands is given below:

cmd - to issue the command to the component. Returns the command identifier. May be started in background form with trailing "&" as a last parameter. The result of cmd when it rejects the request (e.g. component is disconnected) is -1. NOTE this circumstance, since some scripts may be based on issuing the command once previous is over! Since "-1"-result does not cause an error situation in SV (no scenario script interruption is done), this may initiate an infinite loop of cmd-calls (the SV console will be full of messages "Attempt to send a command to disconnected component"). Also, do not forget to enclose the string-type parameters in double-quotes: e.g.:

```
cmd CIRC get longitude latitude
set l [CIRC longitude]
cmd TLSP set longitude="$l" latitude="[CIRC latitude]"
```

is_cmd comid - check that the command with the identifier *comid* (returned by cmd call) is still under execution (concerns evidently the cmd started with "&". Others return when execution is over.) Example:

```
set mass_comid [cmd MASS RUN &]
...
if {[is_cmd $mass_comid]} then { <something is still under way> ...
} else {
<something if command is already over> ...
}
```

wait_cmd comid [comid [...]] - waits until any of the command(s) with given identifier(s) are over. When one of provided commands finishes - return its comid. `wait_cmd -1` returns immediately as well (see cmd). Example:

```
set mc [cmd MASS RUN &]; set dc [cmd DIMM run &]
...
wait_cmd $mc $dc
if {![is_cmd $mc]} {<something for MASS-ready state>}
if {![is_cmd $dc]} {<something for DIMM-ready state>}
# Note that at least one of above two conditioned lines will execute!
```

wait_sec delay [addlog] – to delay execution of the scenario by the *sec* seconds. This does not freeze the SV application updating it once a second. The optional boolean parameter `addlog` may be set 0, if one wishes to skip the addition of the log-record (e.g. if delay is frequent and short).

component **parameter** - returns the value of the parameter (CFG or returned by the GET-command) of the component. In other words, there is such a command for all components which are mentioned in CFG. Request of non-existent parameter causes the scenario interruption on error. Example: see `cmd`-examples.

add_log record – add the record in the log.

initialize "*components list*" – a macro of INIT-commands where them are given the all components altogether (INIT &) and then the macro waits for completion of initialization of all components before return of control. For a single component, the command is equivalent to `cmd COMP INIT`. Note that some components may be initialized with parameters (i.e. like `INIT param=value param2=value...`, see `circ.tcl`) – they cannot be thus initialized with `initialize`.

stop_park "*components list*" – a macro of STOP NOW and PARK & commands where them are given to all components and then the control is given back to caller when all components get status PARKED.

Monitor of circumstances also "knows"

start_obs - start observations scenario (when "good"). E.g.:

```
cmd CIRC get hsun
if {[CIRC hsun]<-12} {
    # Navigation twilight already:
    start_obs
}
```

stop_obs - stop observations. E.g.:

```
cmd METEO get cond
if {[METEO cond]!="GOOD"} stop_obs
```

is_observations_now - is the observations scenario under execution now. Example:

```
if {[METEO cond]=="GOOD" && [CIRC hsun]<-12 && ![is_observations_now]}{
    start_obs
}
```

An example of the observations scenario – `obs_massdimm.tcl` – is documented in the appendix.

Handing errors. As it was stated above, SV takes care of handling the errors which occur while running the scenaria or reported by components in their replies. A normal reaction for fatal errors happening with components (such as a loss of connection, fatal error status etc., see Section E.5) is shutting the component down (parking, if possible, and disconnecting), then – terminate SV or continue, depending on the `optional` parameter of this component.

Meanwhile, sometimes it is more vise to try to handle the situation more flexibly in order to make the system performance more stable. For this, the scenario (monitor or observations) must provide the procedure:

```
proc error_handler {code comp} {
# parsing code and comp-onent parameters...
...
# if handled successfully:
return 1
# else -- unfortunately:
return 0
}
```

Such a procedure must return 1 ONLY if the error of a king `code` from the component `comp` is indeed expected – for example, due to imperfections of its code implementation. See the codes in Section E.5. It is evident that there is no sense to return 1 in some cases: e.g. when the connection is lost (i.e. code==ECMDLOS, ECMDLOW, ECMPDSC), because the next error will occur immediately. Normally, successfully handled are only the ECMPFAT errors, although it is useful sometimes (at developing stage) to report the errors only and to return 1 in any case.

4 Editing the SV configuration file

The SV configuration file name `sv.cfg` (the file resides in the same directory as `sv.tcl`!) is the only hard-coded value of SV. All other settings are made by assignment of respective parameters in this file. Sometimes, `sv.cfg` is a link to the currently active SV setting file.

SV composition is described below and in `sv.cfg` sample as well in the comment lines (starting with #-character).

The parameters of configuration stand in SV as either parameters of SV itself (in this case they are referenced in SV source text as `SV(param)` and in scenaria as `[SV param]`) or parameters of a component - e.g. `[CIRC latitude]` in a scenario or `cmp(port)` in `sv.tcl`.

This division is made with help of so called *sections* in SV configuration file. Beginning of a first section is the beginning of the file. This is the section of SV parameters. They (`tmout, oscen` etc) last till the first declaration of a next sections: "`component name`" on a separate line. This declaration starts the section of the component "`name`" and simultaneously declares that the component program "name" exists in the system and must be connected on start-up using the parameters specified below.

A number of parameters MUST reside within `sv.cfg` in order SV to work correctly. These are the parameters `oscen` and `cscen` of SV and the parameters `port` and `ident` for each component. Their existence is checked during SV startup and error occurs if any is missing. A number of other parameters are also needed but defaulted within SV if not found in `sv.cfg`: these are `tmout, revive_time, interactive, start_monitor` and `emergency_sys` in SV section , and parameters `host` and `optional` in each component sections. Since version 0.20, the parameter `obs_comp_list` is obsolete (due to explicit initialization/parking of components in scenaria, see below).

Finally, a tool for manipulating the observations scenario (only) interactively is provided in a form of a separate GUI: a window with the arbitrary number of buttons. This window appears when Observations scenario is started and disappears when it is stopped. The button action is specified in SV configuration file as a line: `button1` *command in scenario context*; there may be any number of lines like this: `button2 command2`, `button3 command3`... Normally, these commands are simple: assignment of some control variable or calling of a scenario script procedure. Complex commands are not recommended. Usually, such a GUI control is used while developing the system and related observations scenario; when the system is run finally as non-assisted, the respective button parameters are removed from `sv.cfg`.

Below we give a short explanation of a use of SV parameters:
SV parameters:

cscen – file name of a Circumstance monitor scenario script.

oscen – file name of an Observations scenario.

tmout – the time in seconds to wait for a reaction of a component on a newly given command. Default is 10 sec.

revive_time – time in seconds after which the SV is self-restarting after shutting down upon a fatal error occurred with a component while running scenario. This is made in a (maybe fictious) hope that after some time the problem may disappear itself. Defaulted to 0, i.e. do not restart.

emergency_sys – the system command or the name of a system script which is executed upon occurrence of a fatal error before termination of SV. Normally it is shaping out and sending the problem report by e-mail. Defaulted to simply typing "SV termination!" on a local console.

interactive – logical (0/1) variable specifying the way some error situations are treated. If 0 (non-interactive), SV shuts down upon any unhandable fatal situation calling `emergency_sys` before; if 1 (interactive), the message is output in a graphic window allowing some bypassing actions to be undertaken which is used while debugging the system behavior locally (i.e. in assisted mode). Defaulted to 0 (i.e. non-assisted).

start_monitor – logical (0/1) variable specifying whether to start or not the Circumstance monitor immediately upon startup (i.e. as shown in Fig. 6). Defaulted to 1.

button*i* command – specification of a command to be invoked in Observations scenario when it is running.

Component parameters:

ident – component program identifier (string), returned upon the request GET IDENT (see below).

port – socket port number of a component.

host – name of the machine (IP or DNS name) where component is running. Defaulted to 127.0.0.1 (i.e. local host).

optional – role of a component in a system. If specified as 1, the component is parked and disconnected upon occurrence the fatal error with it and the system proceeds running. If not specified or given as 0 – occurrence of a fatal error with the component causes the system to shut down. It is fully due to the scenaria writer to decide which component is optional and which is the mandatory, i.e. the one without which the system performance is not possible at all. Defaulted to 0 (i.e. mandatory).

All the rest parameters are suited for the scenaria needs only and ignored by the SV code itself. Note that there is NO principal difference where to put these parameters – in a section of SV or of some component; it's a matter of taste and free logics. A good habit is to put the personal parameters of components (which control their usage or which are set in them with

SET command) in the section of respective component. Note that similar component parameters may have the same name for different components. In principle, the user is free to hard-code all the parameters in scenaria scripts and to leave only the obligatory parameters in `sv.cfg`.

Example of usage of SV configuration parameters in Circumstance monitor scenario is given in sample `circmon.tcl`.

5 Starting the system

The SV package should be unpacked in some directory, normally it is `/usr/src`. No root privileges are demanded for SV running. So, its useful to change the permissions for the user going to start SV (e.g. `mass`) to be able to write in the SV directory. Also, the log-files `*.log` and temporary files (`tmp.*,.tmp`), if present, and SV configuration files `*.cfg` must be writable or owned by this user. `sv.tcl` and the component scripts such as `meteo.tcl`, `circ.tcl`, `objm.tcl` etc. must have execution permissions.

The component programs apart from those residing in SV directory must be installed separately and must be accessible via network. Their IP addresses (127.0.0.1 for the same machine as SV runs on) and port numbers must be written in SV configuration file `sv.cfg`. Note, that port numbers on the same machine cannot be shared! All component programs should be started BEFORE SV starts and continue to be on-line all the time the system works. Once the connection is lost, SV stops working and disconnects the component. If the parameter "optional 1" is not set in the respective component section of SV, the SV terminates (or gives the respective message if run in interactive mode).

You may use `client.tcl` *port* [*host*] script for connecting to the particular component program to check how it works and how it is seen to SV through network. The commands there are enumerated automatically, so the first input word from the console should be the command keyword (INIT, RUN etc).

Finally, shape out the `emergency_sys` parameter in `sv.cfg` to have the proper error handling command ready. Try to initiate the error situation (disconnect or shut down one of components) and see what will happen: somebody should receive an e-mail in reserved address with the fragment of SV log file if your `emergency_sys` is set to running of the provided `sv_error.sh` script.

Then start Supervisor from the current directory:

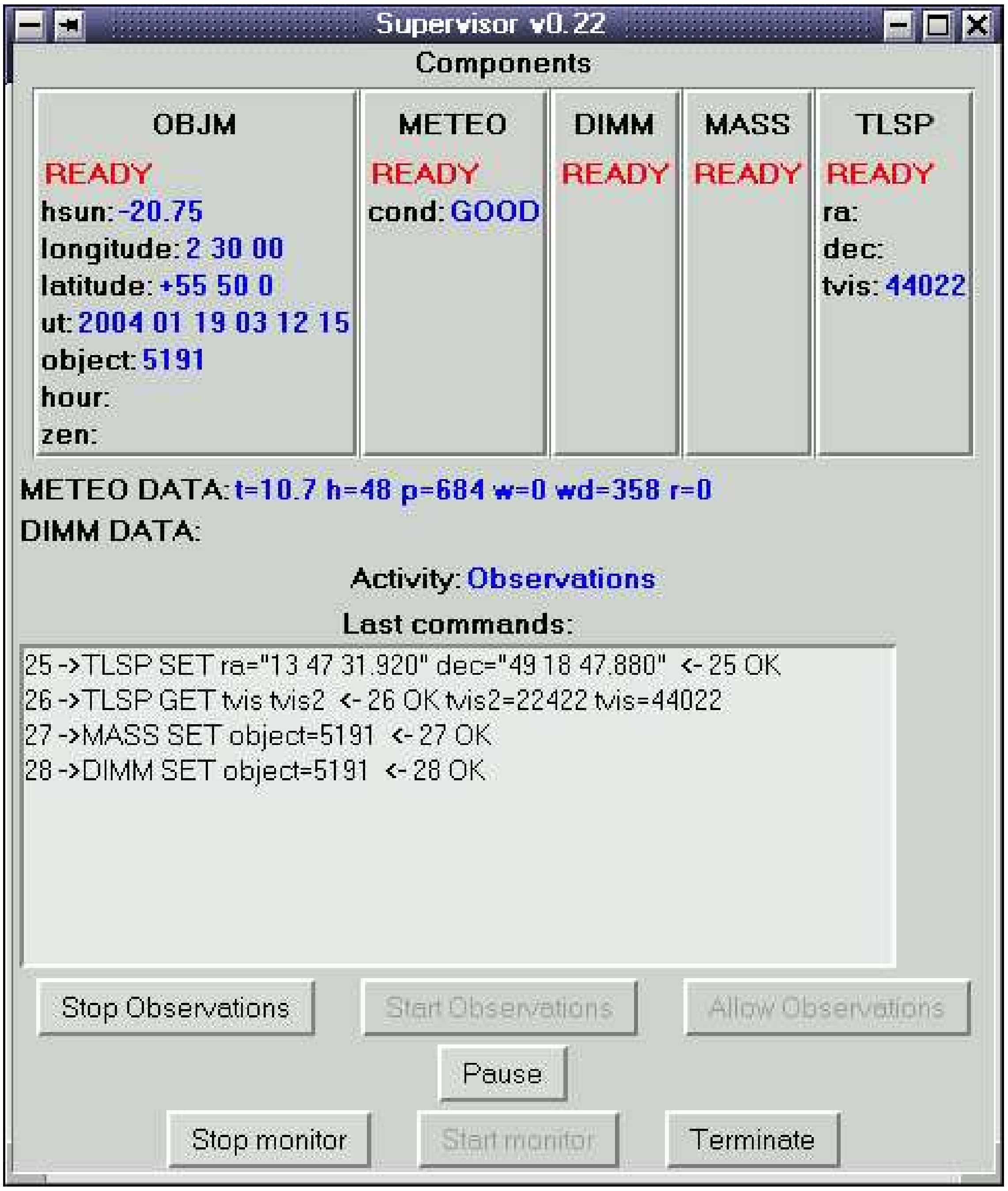
```
$ ./sv.tcl
```

The window like shown in Fig. 1 appears in the screen. The history of commands exchange appears on the console and (somehow shortened, command-reply pasted) - in the SV window panel. Log-file is created with the name YYMMDDsv.log, where YYMMDD is the *evening* local date of the night (it is given by the moment 12 hours before).

The circumstance monitor scenario will be started automatically and may be stopped by the [Stop monitor] button. When monitor (or you – via [Start observations] button) commands to start observations, the observation scenario script with related communication with components will start. To stop observations or monitor – press the respective Stop-button. When finished completely – press [Terminate].

Experiment with a trial starts of SV using the manual Start observations (from GUI button). Check that system starts initialization and everything goes Ok.

GUI control buttons



```
25 ->TLSP SET ra="13 47 31.920" dec="49 18 47.880"  <- 25 OK
26 ->TLSP GET tvis tvis2  <- 26 OK tvis2=22422 tvis=44022
27 ->MASS SET object=5191  <- 27 OK
28 ->DIMM SET object=5191  <- 28 OK
```

Figure 1: SV graphic window

Component names : these are in practice the buttons which pop up the menu with following items:

- **Command** : opens the dialog for command exchange with the component.
- **Connect** : connects the component if it was disconnected. Practical for the manual commands exchange only when no scenario (monitor or observations) using a component is active
- **Disconnect** : closes the socket channel of the component. All the commands sent after this to component are ignored resulting in a respective log message only

Start Observations : calls the procedure `start_obs` of SV. If the observations starting was disabled, enables it first (that's a difference with the starting of observations from the scenario)

Stop Observations : calls the procedure `stop_obs` of SV and disables starting of observations by the circumstance monitor.

Allow Observations : resets the observations starting protection set by Stop Observations button. Observations MAY then be started by the circumstance monitor

Pause/Resume communication with components: delays the sending of the commands to the components until the repetitive pressing of the button. After pressing the button, the scenaria are executed until the control reaches the next `cmd` call. When Resume button is pressed, the delayed command(s) is sent as normally. Component Commands dialog communication and Terminate button are not blocked in the paused state.

Start monitor : starts the circumstance monitor scenario (which is started automatically upon SV startup procedure)

Stop monitor : stops the execution of the circumstance monitor scenario until next its start by the button

Terminate : stops observations parking all the components, disconnect all components, remove GUI and exit the program. The same as a small cross-button of the SV's window (upper-right side normally).

Appendices

A Command list for the communication of the system component programs and Supervisor

Introduction

This document concerns the further development of the software for the project at the upper logical level. Here we consider the requirements for the component programs (like MASS, DIMM and Telescope driving programs and other secondary services) which should be fulfilled for their successful operation under the management of the Supervisor program. The communication protocol was fixed in the previous report (see Appx."Inter-program communication protocol") while here we specify what in particular is needed to be supported (commands and all parameters which may accompany them) in the device programs within the current project.

Currently, the following component programs are expected to take part in the scenario of Supervisor:

TLSP – telescope driving program: pointing and corrections

MASS – detector 1 program

DIMM – detector 2 program

CENT – guiding unit giving the current star offset from the center

CORR – position correcting unit (normally unified with TLSP)

DOME – dome controlling program (may be unified with TLSP)

OBJM – object manager program and astronomical circumstance computer: sun height, Luna coords..

METE – meteo service program

Changes of the philosophy of the Supervisor, compared to the previous Report, are reflected in the separate document "The construction of the supervisor program for automated observations" (Shatsky, 2-7 July 2003).

Considering the protocol itself, three issues should be stressed with respect to the distributed protocol given in the Report 1 (Appx. C):

1. The message strings are terminated with newline character. This is introduced in order to resolve two messages arrived one after other before the program is able to read the socket buffer

2. Command identifier (the first field in the message) is no longer the arbitrary unique word, but (in the current MASS/DIMM project) the unsigned integer number varying cyclically as 0, 1, 2,..., 65535, 0, 1 etc. This space of allowed values should be enough for any feasible set of commands active in the system for a given moment.

3. Parameters use in commands and replies is made more determined. For example, the parameter cannot be replied without request (except for system-specific special parameters like STATUS, see below) and cannot be requested and assigned simultaneously (intermixture of SET and GET commands is avoided).

The up-to-date version of the protocol is given in the appendix to the current document.

Management of the Supervisor from external is not possible. The only way for intervention in SV work is a ssh login to its machine and sending it a signal (TERM) to terminate correctly. This will cause abortion of observations, parking all components and termination of SV program. Then some changes may be introduced in the configuration of SV and its algorithms and SV may then be restarted again.

The current state of SV will be visualized to external by supporting the special part of SV which will monitor the state of components (their status and data report requested by GET DATA command, see below), current object and coordinates including. The respective html-page will be generated automatically from these data upon request from external client for which the separate (server-type) socket will be created by SV.

In brief, the system state and some data results will be seen to external world, but no way to affect the system performance will be provided except for specially permitted login to the SV machine for maintenance.

Other issues of the system configuration (components distribution in different machines, time synchronization etc) will be considered elsewhere and are not the subject of the current commands-definition document.

A.1 Commands acknowledgment

Being fed with a new command, the component must distinguish the commands with purely logical actions (value setting or getting) and the ones which demand some practical job to be completed. Formers normally take a little time to execute since neither extensive calculations nor intensive physical movement or long data accumulation are requested. These commands are executed immediately and the acknowledgment "OK" with some optional supplied parameter-value pairs is sent back to SV.

The latter (intensive) commands – RUN, INIT and PARK – normally take a significant time for execution; their reception and execution beginning is usually acknowledged as "OK STATUS=BUSY WAIT=*twait*" where *twait* is a maximized estimation of the execution time of the command. If the component receives the command "GET STATUS" in between the command reception and the command completion, its status is again returned as "OK STATUS=BUSY". When the command is completed, the final acknowledgment is sent together with the status of execution (see statuses list in sect. A.2).

In case the command arrives which requests the state-switching action while the component is in the incompatible state (reflected by status), the error is replied:
ERROR STATUS=*status*.
Examples are any commands except for STOP NOW and GET STATUS arrived before a previous command was not completed (reply is ERROR STATUS=BUSY); RUN, STOP in the parked state (reply is ERROR STATUS=PARKED). The commands STOP NOW and GET STATUS are accepted in any state, the former causes no effect if there is no job to stop (status is not BUSY).

The maximal time which is needed for any component to react to logical ("short-term") commands should be estimated for any system, maximized by some factor (say, 3) and set as a *timeout* duration in the supervisor needed to detect the abnormal states of the components when they ignore the commands.

It should be stressed again that since the supervisor needs to know the state of the component after completion of each long-term or state-switching command, the components are obliged to report their status while sending the final completion acknowledgment for state-switching commands, namely – RUN, STOP, INIT, PARK, FREE reported as

```
OK STATUS=BUSY|READY|PARKED|LOCAL [WAIT={\it Twait}]
```

and any other (GET/SET including) which caused the error-state of the component replied as

```
ERROR STATUS=ERFAT|ERSYN|ERANG
```

Example of communication log-file (each line being the direction of a message, recipient name and a command):

```
->1000 MASS RUN
<-1000 OK STATUS=BUSY WAIT=61
...
->1020 MASS GET STATUS
<-1020 OK STATUS=BUSY
...
<-1000 OK STATUS=READY
```

In the sections below we list the commands which should be supported by the components. The expected reply is given for each command for the case of success. In case of the fatal error occurred during the command execution, the answer will be "ERROR STATUS=ERFAT"; for errors of the command interpretation replies are either "ERROR STATUS=ERSYN" (for unrecognized command syntax, parameter names etc, including the full character mess) or "ERROR STATUS=ERANG" for the case of the command parameter value out of allowed range (e.g. TLSP pointing to inaccessible direction on the sky or object identifier not found in the detector component catalogue).

Note: for the state-switching commands INIT, FREE and PARK the command repetition is not prohibited. If the component is already in the needed state, it replies immediately as OK STATUS=*state*. E.g.:

```
->1030 TLSP PARK
<-1030 OK WAIT=100
...
<-1030 OK STATUS=PARKED
->1033 TLSP PARK
<-1033 OK STATUS=PARKED
```

The scheme of states and transitions for component programs is shown in Fig. 2. These general rules should be obeyed by any program which conforms to the protocol.

A.2 General purpose commands

Here we give a set of commands with parameters which should be supported by each of the program components interacting with Supervisor.

GET IDENT - return the component identification. This value must coincide with the component parameter IDENT in the configuration file. Since multiple devices are planned to be used at different sites, the safe identification is seemed to consist of the program name, program version and the hardware device ID. E.g.: MASS(IDENT)="turbina v2.01 unit01"). This identification is also expected to be present in the output data files of detector components.

Reply: OK IDENT="*ident*"

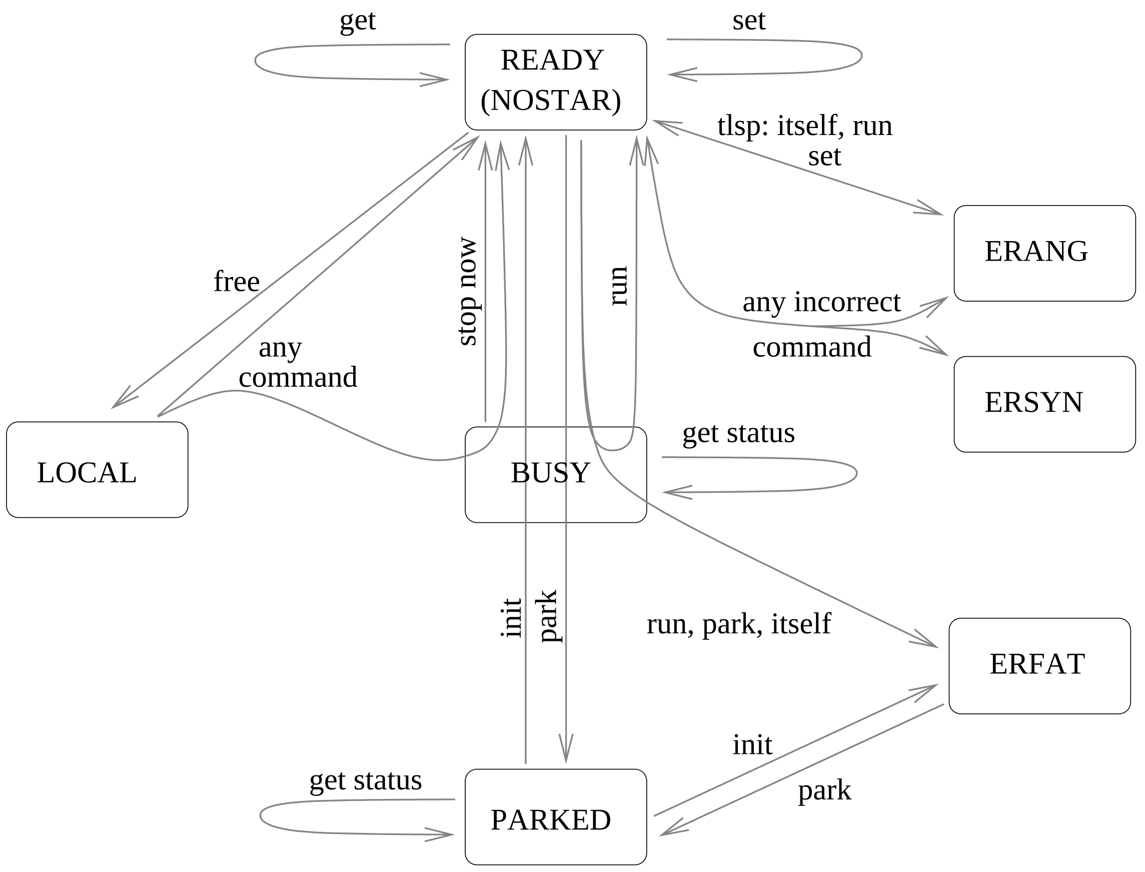


Figure 2: States (statuses) of the component program and transitions between them. Along transition arrows the commands are shown which cause such a component transition.

INIT - self initialize to bring the device in the working (online) state after PARKed state.

For TLSP this means that only the command RUN RA=... DEC=... is needed to start observations, i.e. sidereal tracking will be switched on after pointing (it does not matter to SV whether it is on or off in between INIT and RUN commands).

Replies: OK WAIT=*Tinit*, then OK STATUS=READY

RESET - clear the message queue in the program (no acknowledgment!). This command is reserved as a possible future mechanism of problem resolution using a special mode of socket connection.
Not realized in a first implementation of a system.

Reply: no

GET STATUS - return the status of the program.

The statuses may be following:

PARKED - component in the parked (hardware off) state

LOCAL - component is under the local console control

READY - ready to perform next command (normal state after acknowledging of the previous command with **OK**).

BUSY - command is under execution

ERSYN - error of syntax parsing: unrecognized command

ERANG - supplied parameter is out of allowed range of values. For telescope, this is also the state when telescope has tracked to the limit of position and stopped waiting for the next pointing

ERFAT - unrecoverable error occurred making impossible further work of the component. Details are in local program log file.

There is also a special status for detecting components like CENT or scientific detectors (MASS, DIMM):

NOSTAR - object is not found or sensed within field of view; no valid measurement results may be or are obtained

Reply: OK STATUS=*status*

GET DATA - return the unformatted string with latest obtained output data. This command concerns only the data-producing components, e.g. detectors, centering unit (star offset data) and meteoservice. These data are supplied to the special part of the Supervisor which provides the monitoring of the system via the WWW interface.

Reply: OK DATA="*data record up to 1024 symbols*"

FREE (obsolete) - local control (standby state) activation (if there is one) until a next command comes.

Reply: OK STATUS=LOCAL

Note, that this status may be returned only by the command FREE, since any after-going GET STATUS would bring the component back online. If the component was running some action started locally in the moment of this next GET STATUS, the status BUSY will be returned, until the action is completed or STOP NOW command comes. In any case, no further local action requests is processed. This command is obsolete, since in practice the local controls allow to activate themselves even if SV is connected. So, after such an activation the control should rest local until SV requests the next action.

PARK - shut down own services, switch off the high voltage, close the doors or whatsoever to prepare for sleeping and go asleep.

Reply: OK WAIT=*Tpark*, then OK STATUS=PARKED

QUIT - implies PARKing (if not PARKED status) with a reply as stated above, and termination (with closing the socket connection) after **at least** 1 second. This command is added later and is essential for some robotic systems which do not work in day-time.

In following sections we consider the commands specific to each of the components by their parameters and/or actions required.

A.3 TLSP: Telescope driving commands

The component TLSP must accept the following specific commands. Note, that the coordinates involved in the communication are the real **astronomical coordinates**, not the instrumental ones! In this respect, maintaining the correct time reference system in Supervisor and TLSP component is the crucial part of the correct system performance.

Telescope component commands would be rather more concise if there was no need for SV to check the visibility conditions for a given object, taking into account two possible orientations of the tube relative to the mounting in many telescope mountings. Thus, some commands were introduced to allow the SV to see, how long the selected star may be tracked by the mounting in a given (current or changed) orientation. So, some intellect is required to be supported by TLSP component program to provide these simple but mounting-specific up-to-date measures. However, if there is no way to implement this functionality in the TLSP program, it should respond ERROR STATUS=ERSYN to such requests. Thus, the SV observation scenario should be adapted correspondingly.

SET RA="*hh mm ss"* **DEC="**$\pm$*dd mm ss"* - set the coordinates for future pointing. This is used to check the accessibility conditions in different orientation of the tube with help of GET TVIS TVIS2 command. No status is changed.

Reply: OK

Note: no checking of the accessibility is performed. After-going RUN without parameters may result in ERROR STATUS=ERANG if preset coordinates were unavailable.

GET TVIS TVIS2 - get the time of visibility in seconds for currently preset coordinates (by previous SET RA... DEC... command) or for actual coordinates if no coordinates were preset after the last pointing. Here TVIS is the visibility duration in current orientation of the tube, TVIS2 - in opposite orientation of tube.

Reply: OK TVIS=*tvis* TVIS2=*tvis2*.

Note: if the object is not accessible in one of orientations (or both) or no coordinates were pointed or preset before, respective TVIS or TVIS2 is/are returned zero.

RUN [OVER] - point to the coordinates preset by foregoing SET RA=... DEC=.. command. If no SET-command was given, ERROR STATUS=ERANG is returned.

Optional parameter OVER means changing the orientation of the tube to the opposite while pointing to the preset coordinates (see command GET TVIS TVIS2).

Replies: OK WAIT=*Tpoint*, then OK STATUS=READY
or: ERROR STATUS=ERANG if the requested coordinates are currently inaccessible in the given orientation (no OVER) or in the changed one (if OVER switch is supplied).

RUN RA="*hh mm ss"* **DEC="**±*dd mm ss"* **[OVER**] - point to the given coordinates.

Optional parameter OVER means changing the orientation of the tube to the opposite while pointing to the given coordinates (see command GET TVIS TVIS2).

Replies: OK WAIT=*Tpoint*, then OK STATUS=READY
or: ERROR STATUS=ERANG if the requested coordinates are (currently) inaccessible.

Note, that for german-type and some other types of mountings, there are some limitations for tracking regarding the orientation of the tube relative to the mounting and object height above horizon. If a limit of tracking is reached, the telescope stops and replies to the next GET STATUS with ERROR STATUS=ERANG. This is the only way for SV to know about this state. TLSP must start reporting ERROR STATUS=ERANG to GET STATUS at least **one munute** in advance to actual stopping of the tracking motor.

GET RA DEC - get the current **actual** (not preset by SET RA=... DEC=...!) position of the telescope.

Reply: OK RA="*hh mm ss"* DEC="±*dd mm ss"*.

RUN DRA=*dh* **DDEC=***dd* - apply the correction in right ascension and declination directions. Both *dh* and *dd* are measured in seconds of arc in the sky plane, i.e. the actual change of a hour angle is $\sec(DEC) \cdot dh/15$. The speed of correction is selected by the telescope software, probably depending on the value of the correction.

Reply: OK WAIT=*Tcor*, then OK STATUS=READY
or: ERROR STATUS=ERANG (in case of too large corrections or ERANG (stopped) state of the telescope).

A.4 MASS/DIMM: Detector commands

The following list is supported by any detector device program disregarding its specialization. Both MASS and DIMM devices support the same command set.

SET OBJECT="*nnnn***"** - informs the program that the telescope is pointed to the object identified in the program's target catalogue with a key *nnnn* (in MASS and DIMM *nnnn* is accepted as a Bright Stars catalogue star number).

Reply: OK
or: ERROR STATUS=ERANG (misidentification of the object)

RUN - start a loop of accumulation of one output data piece (accumulation time measurements). Once obtained a data point, stop.

Replies: OK WAIT=*Tmeas* STATUS=BUSY, then OK STATUS=READY;
or ERROR STATUS=NOSTAR if no data may be obtained due to absence of the object in the field of view.

RUN SCEN1 - measure and save the background level for proper reduction of data in the RUN commands (if applicable). SCEN1 is a name of a scenario where the mode (or a sequence of modes) is specified to make the background measurements.

Replies: OK STATUS=BUSY WAIT=*Tbkgr*, then OK STATUS=READY.

The component which does not need the background responds with immediate OK STATUS=READY instead. No check for the clearness of the field is required.

RUN SCEN2 - start measurements of the stellar flux (MASS only). SCEN2 is a scenario where flux measurement (single or continuous – made until STOP NOW comes) is written.

Replies: OK STATUS=BUSY WAIT=*Tflux*, then OK STATUS=READY.

This command is introduced to allow the remote checking of the fluxes in the several apertures of MASS device.

STOP NOW - abort the current accumulation and stop immediately. Current data piece is lost. This is used in rain-drop emergency, or elsewhere. Usually this command is followed by PARK.

Reply: OK STATUS=READY

GET DATA – return the last-obtained results of measurements. Here ”DATA” are replaced with one of SCIND, FLUX, BKGR, X/LPROFILE, INTEGRAL – see Mass User Guide what is provided.

A.5 CENT: Centering unit commands

RUN DRA DDEC - measure and return the measured offset of a guiding (target) star from the center of the field of view in arcseconds. If grabbing of frames is continuously ongoing, return the last obtained offset.

Reply: OK STATUS=READY DRA=*dh* DDEC=*dd*
or OK STATUS=READY DRA=+999 DDEC=+999 if there is no star found.

Note: in some systems, this might be the check for sky clearness for background measurements (see RUN BKGR command in Sect. A.4).

A.6 CORR: Position correcting unit commands

Normally this function belongs to the telescope.

RUN DRA=*dh* **DDEC=***dd* - see the description of TLSP RUN DRA DDEC command.

In most systems the CORR component is an alias of TLSP program, and the algorithms may refer to CORR for the sake of generality.

A.7 OBJM: Object manager commands

GET T1 T2 T3 T4 - compute and return the following moments for the night expressed in UT. T1 is beginning of evening twilight when (restricted mode) measurements may be started, T2 and T3 - begin and end of the ”deep” night (say, sun height is -18), T4 - end of morning twilight. The rule is that T4 is always the nearest moment in the future. The parameters may (in principle) be acquired separately, but this rule about T4 should be fulfilled disregarding which parameter is mentioned. If the evening twilight is continued by morning one (”white night”), then the result obeys the convention: T2=T3=(T1+T4)/2. The polar day request to the parameters considers the first ”white night” in the future.

Time is given in UT in the following format: ”YYYY-mm-dd HH:MM:SS”.

Reply: OK T1=”*t1*” T2=”*t2*” T3=”*t3*” T4=”*t4*”.

GET HSUN - return the current apparent (corrected for refraction) Sun height in degrees (negative in the twilight/night time).

Reply: OK HSUN=*hsun* (e.g. “OK HSUN=-4.6”)

RUN OBJECT RA DEC TVIS - return the parameters of the star for pointing now which is the "best" according to the OBJM’s opinion. Note that the request to either of OBJECT, RA, DEC or TVIS always initiates the search for a best star, so separate requests of these parameters are dangerous and meaningless. Moon position, twilight (hence minimal brightness) are taken into account properly. Here OBJECT is the catalogue number which is identified by the detector catalogues (HR for MASS and DIMM), RA and DEC are transmitted to the telescope for pointing, and TVIS is a time of visibility in seconds during which this star will remain the best choice for observations.

Reply: OK WAIT=*tsort*, then OK STATUS=READY OBJECT="*id*" RA="*hh mm ss*" DEC="±*dd mm ss*" TVIS=*tvis*,
or OK STATUS=NOSTAR if no potential target is visible.

SET OBJECT="*nnnn*" STATE="*ssss*" - inform Object Manager, that the current object "*nnnn*" is either successfully observed ("*ssss*" is "DONE") and needs no pointing for a while, or was rejected by some component ("*ssss*" is "REJECT", e.g. by TLSP pointing restrictions). This causes OBJM to remove this object from sorting by conditions (at least for some period of time).

Reply: OK, or ERROR STATUS=ERANG

A.8 DOME: Dome driver program commands

RUN DOME=OPEN - open the dome.

Reply: OK WAIT=*topen*, then OK STATUS=READY

RUN DOME=CLOSE - close the dome.

Reply: OK WAIT=*tclose*, then OK STATUS=READY

GET DOME - ask the state of the dome (opened/closed)

Reply: OK DOME=OPENED, or OK DOME=CLOSED or OK DOME=BUSY (opening or closing).

Note that the azimuth of dome (for rotating enclosures) is not considered, since the issues of the equipment saving in day- or rain-time are only considered. If dome is a rotating enclosure with a narrow slit, the synchronization of its azimuth is performed by the telescope program and the DOME component will naturally be the alias of TLSP.

A.9 METE: Meteorology service program commands

This component is meant as some interface to the local observatory meteostation service, which requests the needed subset of parameters from the common access place (http or NFS file). Thus, the interpretation of some parameters may not be straightforward. For example, the **T** (temperature) for SV may be assigned the value of some measurement at, say, 5m above ground.

GET DATA - get the current meteo information in unformatted string, where following designations are recommended:

T - temperature in degrees of Celsius at the height of instrument

H - relative humidity in percent

R - rain flag: 0 - no rain, 1 - rain detected

W - wind speed in meters per second

WD - wind direction counted from north to east in degrees

P - pressure in millibar.

Reply: OK DATA="T=*t* H=*h* R=*0/1* W=*w* WD=*az* P=*p*"

Note that this reply format is an example, since SV does not parse DATA replies.

GET COND - get the local conditions in terms "GOOD" for observations and "BAD" for immediate closing the dome based on the own evaluation of criteria for T, H, R, W, and P (see above). This command will evidently be polled much more frequently than GET DATA.

Reply: OK COND=GOOD, or OK COND=BAD.

B Inter-program communication protocol

Here we quote up to date version of the communication protocol which specifies the syntax of the commands sent of the network between the SV and component programs.

Each message transfered via socket connection is a command to perform some action (make something or return some value) or reply to the command.

Message is an ASCII string terminated by the **newline** character. The string has the following structure:

```
COMID KEYWORD [ PARAM1[=VALUE1] [ PARAM2[=VALUE2] [ ... ] ]
```

First word in a string is a command identifier which is aimed to resolve the possible cases of confusion between asynchronous replies to different commands. It consists of either decimal digits or (case insensitive) alphabetic letters. The protocol does not specify the meaning of this identifier; its only property is that it is unique, at least over a long enough time scope of the communication. In MASS/DIMM application, an agreement to use as an identifier the unsigned integer changing as 0,1,..65535,0,1... etc is accepted (this will be a single global command number variable incremented by the command-sending routine).

Second word in a string (all words are delimited by spaces) is a command key word which specifies the type of action which is requested. Keywords list is minimal and their meaning is as general as possible. Details are specified in parameters. The case of letters is not significant. Keywords consist of up to 8 symbols and may include alphabetic letters and decimal digits 0-9 only.

Parameters PARAM1...PARAMn also belong to a reserved list which is more flexible and easily expandable than the keywords list. Optional values (words after a '=' sign) are either decimal floating point numbers or strings. No boolean logic types is supported. If there is no parameter value given, the parameter is a called below "parameter-switch". There must be no space between the parameter name and the '=' symbol and between '=' and the value. String values which contain spaces must be enclosed in double-quote symbols. Example:

```
12 SET OBJECT="Alpha Leo" SPCL=B7V CIBV=-0.11
```

The protocol demands that receiving of every command except for RESET and acknowledgment commands (replies) must be confirmed. This is done by sending an answering acknowledgment command with the SAME command identifier and one of two keywords "OK" or "ERROR" with optional parameter(s). "OK" is reserved for cases when the command parsing was successful and the demanded action has completed without a problem. "ERROR" is for problematic cases: command misunderstood, invalid parameter names, some parameter is our of allowed range or, finally, the fatal error has occurred during the command execution. Which of the problem has resulted in this error, is specified by the STATUS parameter value, which may be appended to the reply. The command confirmation (acknowledgment) is waited for at most TIMEOUT seconds (TIMEOUT is a parameter specified in the SV configuration file which is somehow maximized reaction time of all the programs).

When the confirmation OK is supplied with a parameter "WAIT=<n>", this means that the requesting agent (normally SV) should wait for the completion of the requested action another <n> seconds during which the final confirmation "OK" or another "OK WAIT=<m>" should arrive with the same command identifier. If neither comes, this is a communication failure case.

If the command is expected to return some value of a parameter (other but some general one like STATUS), this parameter name must be mentioned in the command: E.g. GET DATA or RUN DATA cannot result in OK ONE=1 TWO=2, but only in OK DATA="*data*". Backward is not true, since some parameter-value pairs and parameter-switches may be supplied with no need to return any value of the switch (which may be meaningless). For example, RUN BKGR may result both in simple OK and in OK BKGR=bkgr, depending on the system implementation.

Failure case handling. When no answer comes within timeout interval (communication failure) the component state is considered as abnormal, since the information loss on the socket communication level is hardly possible. The fatal error occurred while executing the command is also the abnormal state of the component. These two cases present the emergency situation which is specially handled by the system manager (SV). When the reply is an ERROR message, this is mostly due to incompatibility of the command set implemented in a component and utilized by the SV algorithms. This case also causes the emergency situation which needs to be resolved by the system programmers.

It is possible to introduce the command repeated sending in cases of the command loss (timeout situation), but it hardly currently seems to be promising in troubleshooting. Meanwhile, this option may be activated in future after getting some experience with working SV-driven systems.

Command Keywords list.

INIT - initiate the program service (bring online) to be able to execute the commands from SV. If needed - perform self-test which is specified in the process configuration file. Set the default settings in the system for working with SV.

RESET - clear the message queue in the program. This command is of the highest priority and serves to resolve conflicts in communication. No reply is sent by the program once received this command. The program service (or device) is not reset or re-initialized

PARK - shut down own services of a program and be prepared to be terminated. Optional parameter QUIT means PARKing and termination of the program.

FREE - give the program control to the local user console. This state continues until the next command from SV

GET - return the value of some parameter. Demands a parameter <NAME>, which is a name of requested parameter. Expected reply is "OK <NAME>=<value>". Parameter-value pairs are NOT allowed, only the (not empty) list of requested parameter names is accepted.

SET - set the new parameter(s) in the program. Most usual is setting the new object. In this case, parameters are likely OBJECT=... RA=... DEC=... SPCL=... MAGV=... CIBV=... MODE=... etc, the exact list depends on the program to which this message is sent (E.g., the TLSP program demands normally the RA and DEC parameters only). Switches are NOT allowed, i.e. each parameter name MUST be followed by the parameter value.

RUN - start action. For telescope - point to object (normally, followed by "OK WAIT=..." and then "OK", for detector - start observation. Telescope may point after a single RUN RA=... DEC=... command, i.e. without preceding SET RA=... DEC=... command.

STOP - stop observation after end of current measurement (for TLSP - stop movement immediately). Optional parameter NOW demands to abort the measurement instead of completion.

C Description of the observations scenario obs_massdimm.tcl

The observations scenario, written in `obs_massdimm.tcl` file, is a special Tcl-script and should be run by means of Supervisor program only (version 0.12 or later).

The aim of the scenario is to provide the joint work of the observational system components - object manager OBJM, telescope TLSP, centering device CENT (which is part of DIMM or TLSP), corrector CORR (normally a function of TLSP) and two independent detectors - MASS and DIMM - for astroclimat simultaneous measurements. The mode of observations - the continuous monitoring of the most appropriate object during the time of its good visibility. The guiding (periodic correction of the telescope attitude) is performed simultaneously with measurements; the background level registration is made from time to time by MASS device after the shift of the telescope by some distance and after temporary canceling of the measurements and guiding.

The algorithm is presented in a block-scheme Fig. 3. The structure is expressed by separation of part of code in several procedures (rectangles on a chart with three sections in each one), from which the general algorithm is constructed. The procedure name is given in the middle of rectangle, the upper section list the names of control variables of algorithm (in capitals) and parameters of components (capitals with parameter name in parentheses) from which the procedure performance depends; the bottom section shows the variables affected by the procedure. The control variables drive the conditional transitions in the algorithm. Besides the procedures, the control variable assignments and usage of supplementary small functions (lower-case named) are also shown in block-scheme. Finally, the dashed arrows point to the commands "after <delay-time>" using which the background delayed execution of some procedures or commands is realized. For example, the BACKGROUND procedure sets the BG_NEED flag for own restart after a certain period, while GUIDER restarts itself using after-command with another periodicity.

The procedures themselves, depicted by rectangles, implement in turn the algorithms of more local meaning with different degree of complexity. Their block-schemes are not considered (being quite simple in most of cases) but the principles of work and some details are given below for them.

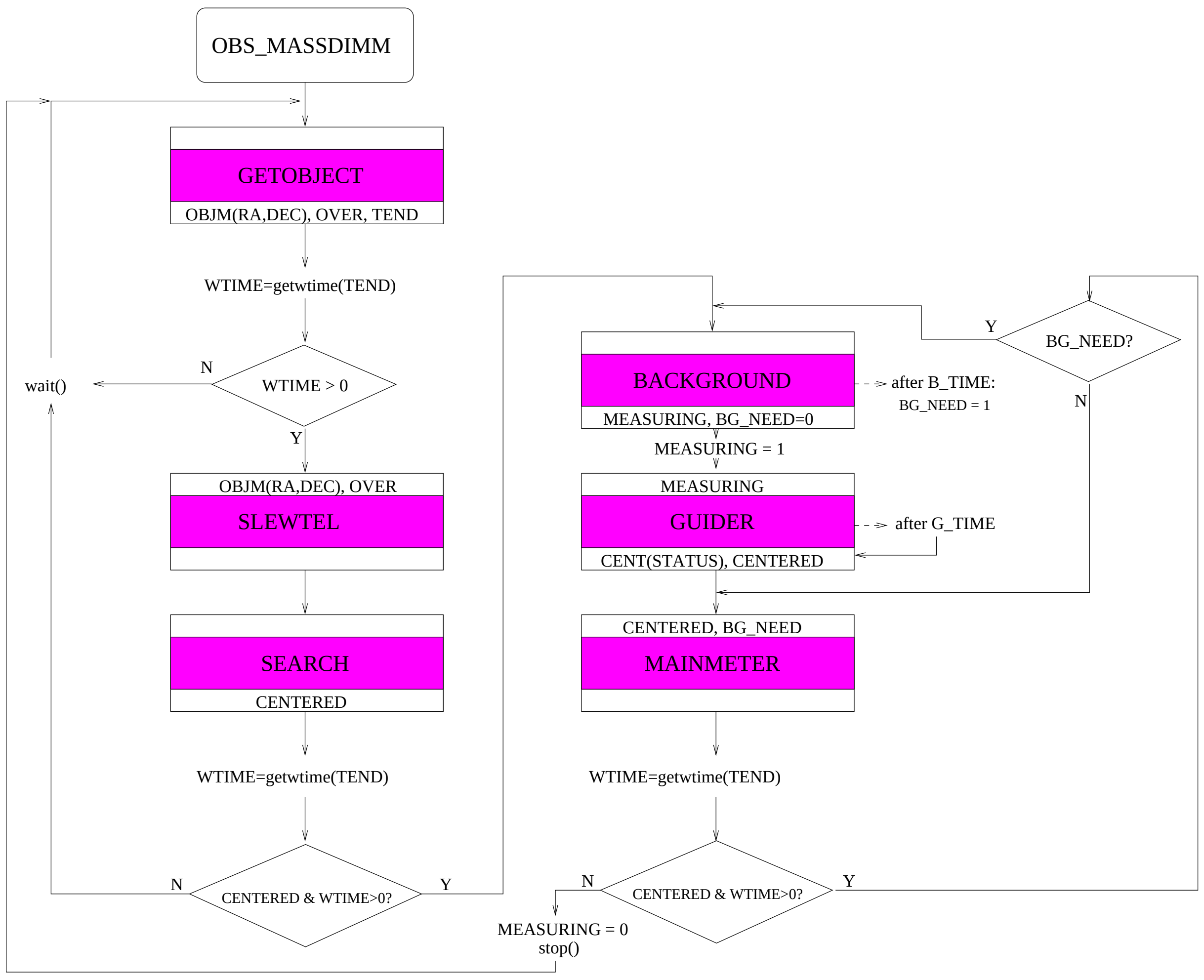


Figure 3: Block-scheme of the algorithm of observations with MASS/DIMM

C.1 Algorithm functioning

The work with any new star is started, obviously, with a selection of an object (i.e. the star itself) - this is GETOBJECT procedure. The accessibility of the star, which is the best according toe OBJM, by the telescope is checked as well, using the current or changed telescope tube orientation relative to the mounting. Normally for fork-type mountings the accessibility is always good. As a result, the needed orientation is selection (normally - the current one) and the limiting clock value TEND till which the star should be tracked is derived (in seconds after 0h of 1 January 1970, i.e. in Unix-seconds).

Using TEND the observations time WTIME is computed. If this is zero (no appropriate star is accessible, ... possible in principle), then the selection of a star is repeated after some time (10 min). In the end of procedure, the object is set in the component programs MASS and DIMM by the command SET OBJECT=<HR-number>.

The telescope is pointed to the selected star (SLEWTEL procedure) and the algorithm of the star search by the square-cornered spiral-like tube movement around the expected position is started (SEARCH) until the star is found or the limiting radius of the spiral is reached. The found star is centered by GUIDER procedure. In case the star is either not centered (wind, or correcting problems) or not found (clouds?) - these two reasons are expressed by a single flag CENTERED - or the centering has consumed all the available observations time (possible as well) - the waiting procedure is again called and algorithm is restarted.

When the star is successfully centered, the measurements begin. First of all the background registration procedure BACKGROUND is called: it shifts the telescope off from the star, sends the MASS the command RUN BKGR and shifts the TLSP back compensating for the possible backlash. After making these perturbations, the setting of the BG_NEED flag again in 1 is scheduled using the after-command. The time after which the flag will be set is determined by the circumstances - the Sun and Moon altitudes. After return from BACKGROUND the algorithm allows the star measurements setting the flag MEASURING to unity.

The GUIDER started afterwards is executed only in case the MEASURING flag is set. It requests the current star displacement from the centering device and, with a negated sign and some alpha-regularization, this displacement is translated to the telescope (correcting device). This is repeated at most 3 times. After this the next restart of procedure is planned after some short time and the procedure exits leaving the flag of success determined - CENTERED. Note that GUIDER is also used by SEARCH.

Finally, after centering, the MAINMETER is started which performs the cycle of the main measurements. It waits until one of detectors finishes measurements and, upon this reads the data string from this detector. Finally, it restarts the accumulation time measurements if the centering is not violated or BG_NEED flag is still reset.

Next, if the time is still left and the centering is Ok, the BACKGROUND, GUIDER and MAINMETER are called in case BG_NEED is already set, or MAINMETER solely is restarted if not. The call to BACKGROUND resets the MEASURING flag which stops the self-starts of GUIDER. Background work is made, MEASURING is set back, GUIDER is revived and measurements are again performed - until the time is over (WTIME==0) or centering is violated due to clouds or wind (CENTERED==0). If one of two occurs, guiding and measurements are stopped and the algorithm of observations is restarted.

C.2 The list of control variables of algorithm

BG_NEED : flag of necessity of background measurements. Set by background command execution scheduled in BACKGROUND. It is the condition to start BACKGROUND explicitly, the latter is not the self-restarted! BG_NEED is reset by BACKGROUND.

CENTERED : flag of success of centering: star is found and is put within the working zone.

TEND : limiting system clock value (in seconds after 1969) till which the object measurements may be continued (see OVER).

MEASURING : flag of ongoing measurements - in practice it's a flag of telescope centered on the star when the guiding and measurements are possible. Reset in BACKGROUND and after the end of object observations time. Set explicitly after return from BACKGROUND.

OVER : flag of necessity to change the telescope orientation while slewing to object. Set in GETOBJECT in case when the time of accessibility of object in current orientation of telescope is less than the time of object visibility according to OBJM (i.e OBJM(tvis)¿TLSP(tvis)), but the opposite orientation results in longer measurements series (i.e. TLSP(tvis2)¿TLSP(tvis)).

WTIME : time left till the end of observations in seconds. Determined by getwtime() call from the current time and TEND.

C.3 Procedures of algorithm

Below we give the list of procedures of algorithm with the brief explanation of each one. The control variables used and/or determined are not listed being shown in Fig. 3 and explained above.

GETOBJECT - requests the object manager the object (number) with coordinates and time of visibility. Determines the necessity of orientation change of telescope. The latter implies that the solution about change is due to the mounting user (i.e. SV); if the orientation is selected automatically by TLSP, the procedure should be simplified

SLEWTEL - points the telescope to the selected object

SEARCH - centering of the slewed telescope on the star just as the star appears in the field of view of centering device (e.g. DIMM). If the star is not found from the first shot, the spiral movement around the central position is made until the star is not found or maximal distance to the center position is reached. In principle, the identification of the found star is possible based on the stellar brightness check, but this is not currently implemented. Centering of the found star is made by GUIDER call (which is not self-restarting in this case since MEASURING flag is not set).

BACKGROUND - stop the guiding and deviate the telescope by some distance. Then - register the background level using MASS. After this - move the telescope backward by double distance and again forward to compensate for back-lashes. The time after which the background registration is again desired is determined by get_bkgr_period() call. After measurement of background the BG_NEED is reset and its setting is scheduled by the background command (after in Tcl) after the determined period.

GUIDER - iterative (at most 3 attempts) correction of the star position in the field of view by reading its shift from CENT and sending it negated to the corrector CORR. Explicit call is distinguished from the self-restart by the argument (default or not). In background (self)restart while the MEASURING flag is not set, neither iterative correction nor self-restart is not performed. The result - star found or not (CENT-result is coded as +999 if not found) and centered (put within "working" zone) or not is expressed by the value of the single control variable CENTERED.

MAINMETER - the main procedure of scientific measurements. Operates with the identifiers of RUN-commands sent to MASS and DIMM detectors by which it waits for end of MASS or DIMM measurements. If one (or both) detector finish its own accumulation time, it reads its data. According to the values of CENTERED and BG_NEED flags, either restarts the ready detector or doing nothing. No explicit iterations or restarts is made - this is done on the global (algorithm) level. Thus, the current version of MAINMETER makes the measurements with two detectors ASYNCHRONOUSLY, with a maximal usage of the observational time. In future versions the modification is possible to implement the simultaneous start of both detectors after waiting for the one which finishes later.

STOP - stops both detector measurements, disregarding their status. Resets the scheduled setting of BG_NEED or start of GUIDER. Used in algorithm itself and in the BACKGROUND procedure.

getwtime() - computes the time left till the end of observations of the current star as (TEND - seconds) or 0 if the result is negative.

get_bkgr_period() - determines the time span between the current moment and next background measurement using following condition: If Sun altitude is more than -20 degree, or Moon altitude is positive but less than 15 degree, then this is 10 min Otherwise this is 20 min. This simple condition specifying the periods of the fast background level changes may be replaced by some more sophisticated functional or, on contrary, by a simple constant.

C.4 Algorithm usage

The variables-parameters of algorithm (such as the periodicity of the guiding cycles etc.) should be written in configuration file of SV (sv.cfg). Also, the name of this scenario `obs_massdimm.tcl` must be set in the parameter `oscen` in the first section (SV section, see main document). Afterwards, the components participating in scenario are configured (MASS, DIMM, TLSP, OBJM and CENT and CORR which are the aliases of DIMM and TLSP, respectively).

After this one may star Supervisor and observe the work of scenario by means of communication logging. For the debugging and control of the scenario performance the special utility `viewlog.sh` is useful.

C.5 Parameters of configuration file which tune the scenario work

Below we list the parameters put in sv.cfg for tuning the scenario performance without touching its file. The parameter name is given in parentheses after the name of section of the configuration file.

SV(wait_time) - time of waiting for "better times" in seconds

CENT(n_try) - number of attempts in GUIDER made to put the star in the central zone

CENT(cent_zone) - the size of circle within which the star shifts are neglected

CENT(work_zone) - work zone is a much larger circle compared to cent_zone, where the measurements are valid

CENT(alpha) - the regularization parameter for corrections (0.9 normally).

CENT(period) - the periodicity of calls of GUIDER in seconds

TLSP(shift_step) - step of the telescope displacement by the hour angle or by declination while moving it along a squared spiral searching for the star

TLSP(shift_max) - the maximal displacement of telescope from the center of a spiral by either of coordinate while searching for the object

TLSP(bkgr_shift_ra) - displacement by right ascension for background registration

TLSP(bkgr_shift_dec) - displacement by declination for background registration

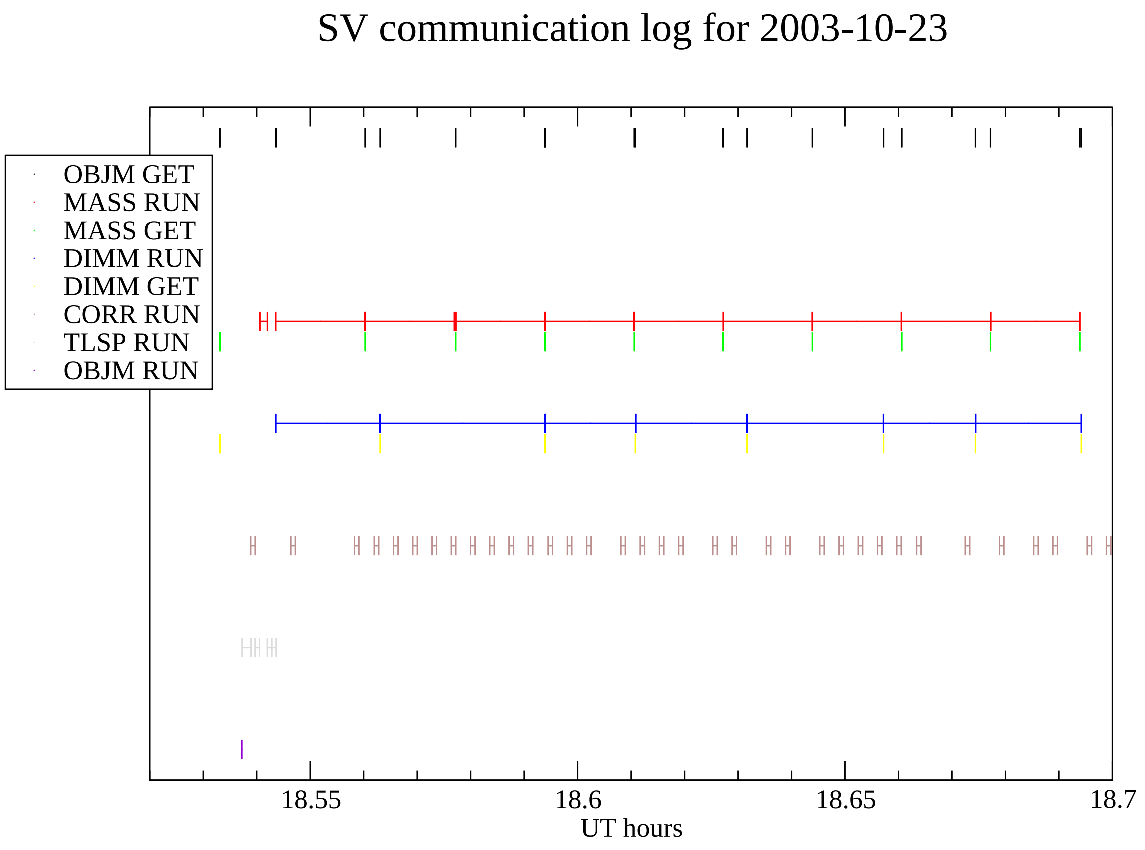


Figure 4: XMGRACE output produced by the viewlog.sh utility

D Visualization utilities for scenaria debugging

The utility `viewlog.sh` is aimed for the visualization of the components communication during the execution of the SV scenaria with help of XMGRACE graphic utility.

It draws the horizontal bars of different colors which reflect the busy-times of the respective components; the color and ordinate of the bar designates the pair of component name and the command keyword which `viewlog.sh` extracts from the log. The abscissa is the UT hours deciphered from the time stamps of the log. The beginning of the bar denotes the moment when the command was issued; the tail of the bar denotes the report about the command execution finished.

In order to configure the utility, one must edit the `viewlog.typ` file and write a line there for each kind of a command which one wants to see on the graph. The first word is the ordinate of the series (in range, say, [1,10]), the second is the component name (real or alias) and the third is the command keyword. Note, that the commands to the same component which differ in parameters only cannot be distinguished in a graph. The color is selected automatically.

Then, the `viewlog.sh` must be started with the name of the SV log-file, e.g.:

```
$ ./viewlog.sh 031023sv.log
```

It will first show the sets for drawing on the console and then start the XMGRACE program (version 5 or later) using the special parameter file `viewlog.par`. Use the menu Plot—Axis properties—x-axis for magnifying some time-span of the interest and adjust the x-axis tick marks correspondingly. Then you will see the picture like shown in Fig. 4

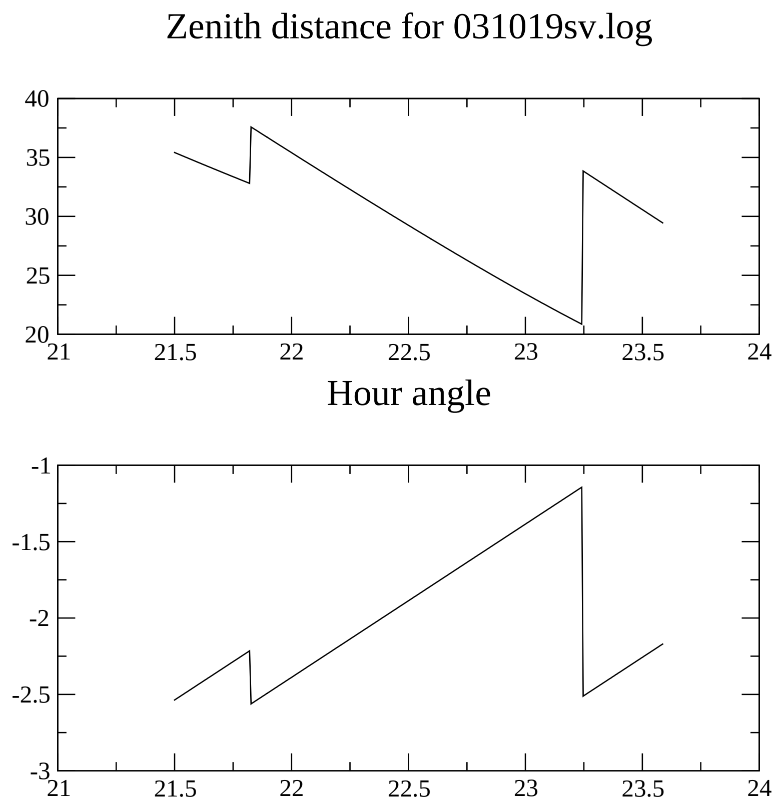


Figure 5: XMGRACE output produced by the viewhaz.sh utility

Apart from `viewlog.sh`, there is another XMGRACE-based visualization utility – `viewhaz.sh`. It helps to see what's a change of the objects hour angle and zenith distance throughout the night. Its only parameter is the name of the SV log file. When started with log-file name, it extracts the "hour=..." and "zen=..." expressions from the log and plots the change of these parameters against the UT hours as shown in Fig. 5. As with `viewlog.sh`, you can tune the X-axis limits to magnify the period of interest.

E Supervisor programming issues (development engineer guide)

E.1 Introduction

In current document we consider the ideas and principles on which the Supervisor programming is based. The general ideas of creation of a Supervisor program and organization of observations with help of SV and associated component programs are expressed elsewhere (see supervisor.txt and Combined MASS/DIMM instrument...Report I., Kornilov et al, 2003).

E.2 Operating system requirements and portability

SV is written in Tcl/Tk interpreted language and can be run in any Linux system with X-windows which has a Tcl/Tk package installed, together with additional freeware package extentions - namely Extended Tcl (TclX) and extended Tk (Tix). Latter packages improve SV functionality but are not obligatory since version 0.21. Since some operating system dependent commands are executed by SV and its associated components (see for example, the current implementation of the Object Manager, OBJM in objm.tcl), the portability of SV to other operating systems where Tcl/Tk also runs is questionable but not impossible. Once a serious reason emerges to do it, the port will be done, since majority of the SV code is a pure Tcl which is platform-independent language.

E.3 Structure of the SV program

SV is considered as an extention of the standard Tcl commands which helps to easily organize the sequencing of the operations with certain components of the observational system with help of the commands exchange in a certain protocol described in commands.tex (*.ps). To make the sequencing life even more easy, three basic processes are completely separated from each other - the monitoring of the conditions, the preparations/cancellation of observations (when the conditions become good or bad, respectively) and the observation actions chain itself. The first and third sequences (scenaria) run in different Tcl scripts and interpreters and have a very thin connecting thread with help of a reserved SV commands like start_obs, stop_obs and is_observations_now. In some respects SV mimics the idea of the VLT Sequencer but uses the own protocol of the commands communication between components and SV and the own strategy of errors handling. The latter is completely separated from the monitoring and observational scripts.

SV text is considered to be completely independent from the task which it is aimed to decide. The application of SV to the particular observational system is made by editing the SV configuration file, from which it knows the list of component programs and some few other parameters which help to communicate with them. All the other information in configuration file serves to the scenaria scripts only. These scenaria are the scripts which describe the desired behavior of the system in details. In some respect, the observations and circumstance monitor scenaria represent the ”supervisor” notion, and the SV program itself is only the background support for this. Of course, in order to have the system working as we want it, the component programs must support the protocol of commands (see commands.tex) and perform the related tasks correctly.

The SV consists of a set of procedures which are glued with each other by the scenaria scripts and by the start-up procedure of SV - main{}. It runs on the event-handling base, i.e. once created the graphic interface, it executes the commands from the separate scenaria interpreters, from the GUI elements (buttons) and processes the information got from the components via open sockets. SV terminates with help of a special procedure terminate{}, which closes the components correctly and removes the SV GUI window thus causing the main{} procedure (see below) to return and exit the program.

The SV procedures belong to the following classes:

the start-up procedures called in main{}: reading the SV configuration file read_cfg{}, creation of the SV GUI (Tk interface) create_window{}, some validation of the scenaria scripts check_scen{}. main{} also calls the multiple-access procedures connect{} and start_monitor{} (see Fig. 6)

the socket-connection service: connect{} with component identification checking, disconnect{};

the command service: the cmd{} utility to send a command to the component, get_reply{} socket channel reading handler to process the reply from the component when it arrives; the helpful scenaria-oriented functions is_cmd{} and wait_cmd{} for checking the status

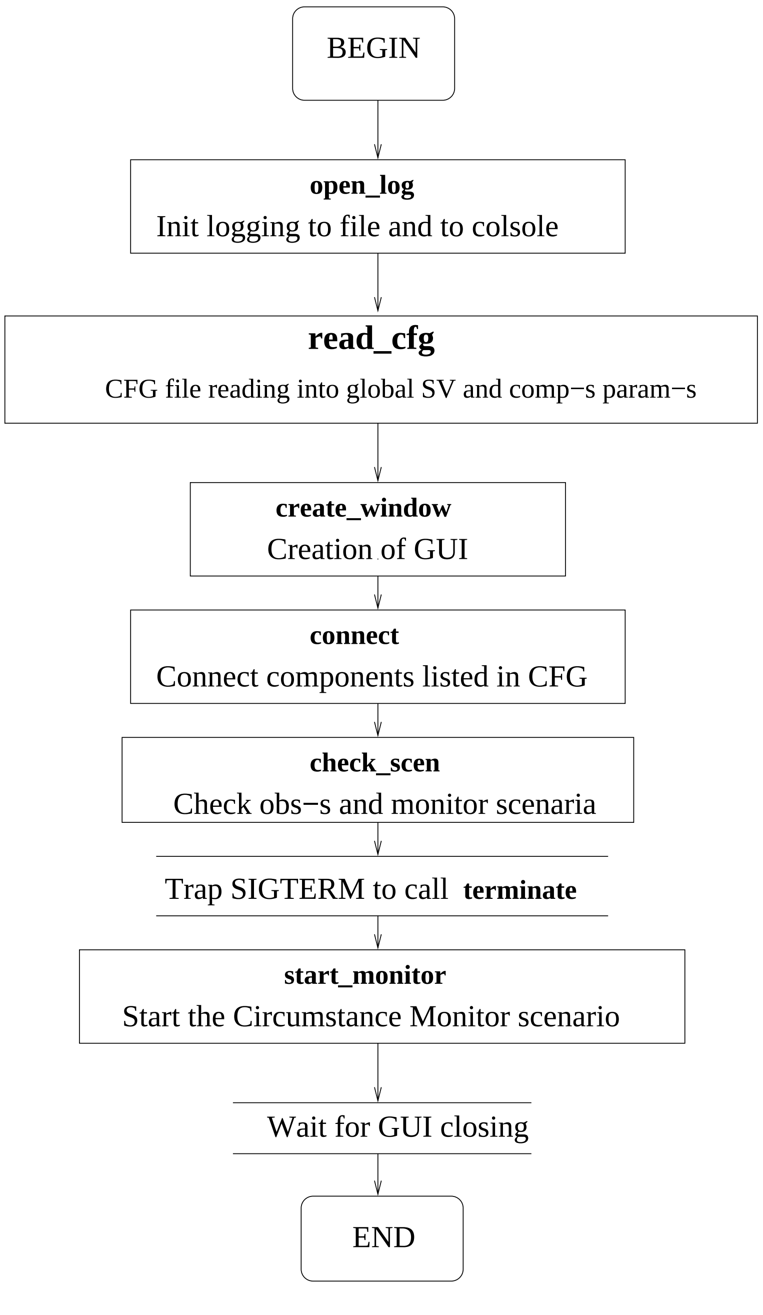

BEGIN
open_log
Init logging to file and to colsole
read_cfg
CFG file reading into global SV and comp−s param−s
create_window
Creation of GUI
connect
Connect components listed in CFG
check_scen
Check obs−s and monitor scenaria
Trap SIGTERM to call terminate
start_monitor
Start the Circumstance Monitor scenario
Wait for GUI closing
END

Figure 6: Startup of SV algorithm

of the component command execution and the runtime-created functions named the same way as components to read the SV and component parameters obtained from configuration file or requested from components themselves;

the batch-like procedures (macros) start_obs{} and stop_obs{} available to the circumstance monitor scenario to start/cancel observations; start_monitor{} and stop_monitor{} for engineer access (GUI control only) to activate/disactivate the circumstance monitor. The separate procedure terminate{} makes all actions in order to park and disconnect the components correctly and cause SV to exit normally. The synchronization of calls to these batches is considered in a section III below;

the batches start..., stop... and terminate use the lower-level utilities: for starting/stopping a scenario: start_scen{} and stop_scen{} which structure is very important for SV performance; macros initialize{}, stop_park{} for sending the INIT and STOP, PARK commands to components when it is needed. They trace execution of these commands and exit only when they are over;

GUI procedures, including those called automatically by Tk "trace variable" commands - update_buttons{}, cmd_dialog{}, cmd_issue{}, update_cmd_array{}. The call to them is organized with help of Tk means in create_window{} startup procedure;

logging facility: open_log{}, add_log{}, close_log{}, time_stamp{};

error-handling procedure problem{} (see section E.5 below).

The start-up sequence is shown in Fig. 6. A number of utilities is called sequentially preparing SV to the normal functioning. Then, before starting the monitor, the signal trapping is made for the usual SIGTERM signal, which is sent by kill program as a default signal to the process. Once trapping this signal, the terminate{} procedure will be called. The same consequence is configured in GUI for the cross-button in the upper-right corner of the window and for the "Terminate" button. Finally, terminate{} is called from problem{} in case some unmanageable errors is encountered. Trapping the signal is the only feature of the extended Tcl (TclX) package which is used in SV. That is why the conditional body is put in main{} for trapping signal only in case the TclX package is activated correctly. This means that SV will work without TclX as well, but it will be impossible to stop SV correctly from another machine.

Most of other procedures in SV are self-explanatory, both by the name, content and by the comments embedded in the procedure texts as well. Procedures are preceded by the comment headers shaped in doxygen style (although no DOXYGEN version still exists for documenting of Tcl projects, unfortunately).

Finally, the SV procedures are all embedded in the single namespace "sv" with help of "namespace eval" command. Since a number of procedures is called from external scenaria scripts, they are exported to the global namespace with help of "namespace export/import" commands. These global names are aliased in turn by "*slave* alias" command in start_obs{} and start_monitor{} batches. In these procedures, the other procedure calls are made with sv:: namespace qualifier. The sv-namespace variables (global to procedures) are referenced in procedures as namespace-qualified. The idea of sv namespace refers to the earliest version of SV when no clear mechanism of the SV internal protection from the other untrusted scripts (scenaria) was introduced. Currently it seems to be excessive, since the interpreter which runs SV code do not run any other scenaria or scripts. Namespace sv will be possibly dropped in future development of SV.

E.4 Starting and stopping the observations and circumstance monitor

Both scenaria are considered to be non-parts of the Supervisor, i.e. treated as (non-trusted) user-written code. User here is a person who is familiar with current observational system structure (component programs usage) and implementation of algorithms Tcl scripts.

Since the code of scenario is user-written, it is considered as a potential source of errors or danger to the system integrity. That's why the scenaria are started in a standalone SAFE interpreters and not in the interpreter which runs the SV code itself.

The interface which is provided to the component programs consists of a few "command"-handling utilities (issuing the command to the component - cmd - and tracing its completeness - is_cmd and wait_cmd) and, for the monitor scenario, also the commands to start and stop the observation scenario (start_obs, stop_obs) and to trace the flag of the observations status (is_observations_now command which returns the fact whether the observations interpreter is running now). Other functions are described in Sec. 3. In addition, both scenaria have a reading access to the parameters of the components and SV which are read from the SV configuration file or retrieved from the components with help of GET or RUN commands with respective arguments. These commands have the same names as components or "SV" and written in upper-case. An argument to these commands is a name of the parameter in interest (lower-case).

Since the starting and stopping of scenaria are not atomic procedures (they have certain and significant duration in time) there is a need to synchronize their calls from different "threads" of the SV performance (observation and circumstance monitor scenaria, GUI, remote program termination). For example, the starting of the scenario by the circumstance monitor may be interfered by another start or stop from the other source, for example - from the button hit on the SV window. Other example is termination of observations: it involves normally the continued parking process of the components and it would be strange to allow to reenter this parking from the other source (e.g. the monitor scenario stopping of observations and the remote SV termination).

Starting of a scenario is made with help of the procedure `start_scen{}` which creates an interpreter, aliases to commands within it and invokes the (hidden) source command with a needed scenario script file. Stopping the scenario is more tricky: it should cancel the evaluation of the scenario not violating the rest functionality of SV and the operated system which demands depends on the particular task. That's why the reserved-name procedure `end{}` is searched for among the known scenario interpreter procedures. If found - it is started hoping that it will stop the observations sequence correctly which is the competence of the scenario-writer. Then the aliases are removed and interpreter is deleted and its variable is assigned zero. NOTE, that removal of aliases is formally excessive (since deletion of interpreter deletes aliases as well), but some machines running Tcl version 8.4 behave strangely: during about 1 sec they continue to execute the scenario commands AFTER the interpreter was deleted! This might be the consequence of some bufferization made inside Tcl code. What saves situation is a still immediate removal of aliases: the running script becomes blind and handless and cannot affect the system components any more.

A number of SV global variables are responsible for controlling the access to these procedures. They are used as flags or semaphores which allow or prohibit the execution of the body of respective procedures. Here below we consider these variables:

```
is_observations_now (1/0): fact that the observation interpreter runs the
obs.scenario
->1 in start_obs\{\} just before starting sourcing obs.scen
->0 in stop_obs\{\} just after obs.interpreter is deleted
```

```
disable\_start\_obs (1/0): fact that starting of obs-s is not allowed
->1 in start_obs\{\} after beginning
->0 in start_obs\{\} before end
->1 in stop_obs\{\} called from the SV window button and terminate\{\}

tclobs ("interp?"/0): obs.interpreter identifier
->[interp create -safe] in start_scen\{\} beginning
->0 in stop_scen\{\} after deletion (in the end)

tclmon ("interp?"/0): circ. monitor interpreter identifier
->[interp create -safe] in start_scen\{\} beginning
-\>0 in stop_scen\{\} after deletion (in the end)
```

Procedures which are involved in process of starting/stopping the scenaria are listed below. "Conditions" specify the ones which should be fulfilled to call the procedure.

start_scen{} : create interpreter for a given scenario assigning the given variable with the interpreter name, creates aliases, sources scenario catching its errors (which stop sourcing thus)

Conditions: no

stop_scen{} : calls the end{} procedure in the scenario (if found), deletes aliases and interpreter itself, assigns its variable zero.

Conditions: interp!=0 && no_interp_in_stopping_list

start_obs{} : start the obs.scenario with help of start_scen{}

Conditions: !disable_start_obs && tclobs==0

stop_obs{} : delete obs.interpreter with stop_scen{}

Conditions: [tclobs exists]

start_monitor{} : start the circ.mon.scenario using start_scen{}

Conditions: tclmon==0

stop_monitor{} : delete circ.mon.interpreter with stop_scen{}. Parking of monitoring components is made in terminate{}, so monitor normally does not contain end{} procedure.

Conditions: [tclmon exists]

terminate{} : call stop_monitor{}, stop_obs{} if under way, stop_park{} for all components

Conditions: non-reenterable (vwait-block of is set to avoid repetitive stop_park{}-ing)

GUI : buttons to call start/stop_obs/mon{} and terminate{},

Conditions: switching of their active-state is made according to the value of the above-presented global variables

The mentioned conditions are verified in procedures themselves and used to activate/disable the respective buttons in GUI. Latter switching is only made to inform the operator about the true state of the program, because the procedures protect themselves from non-contemporary calls.

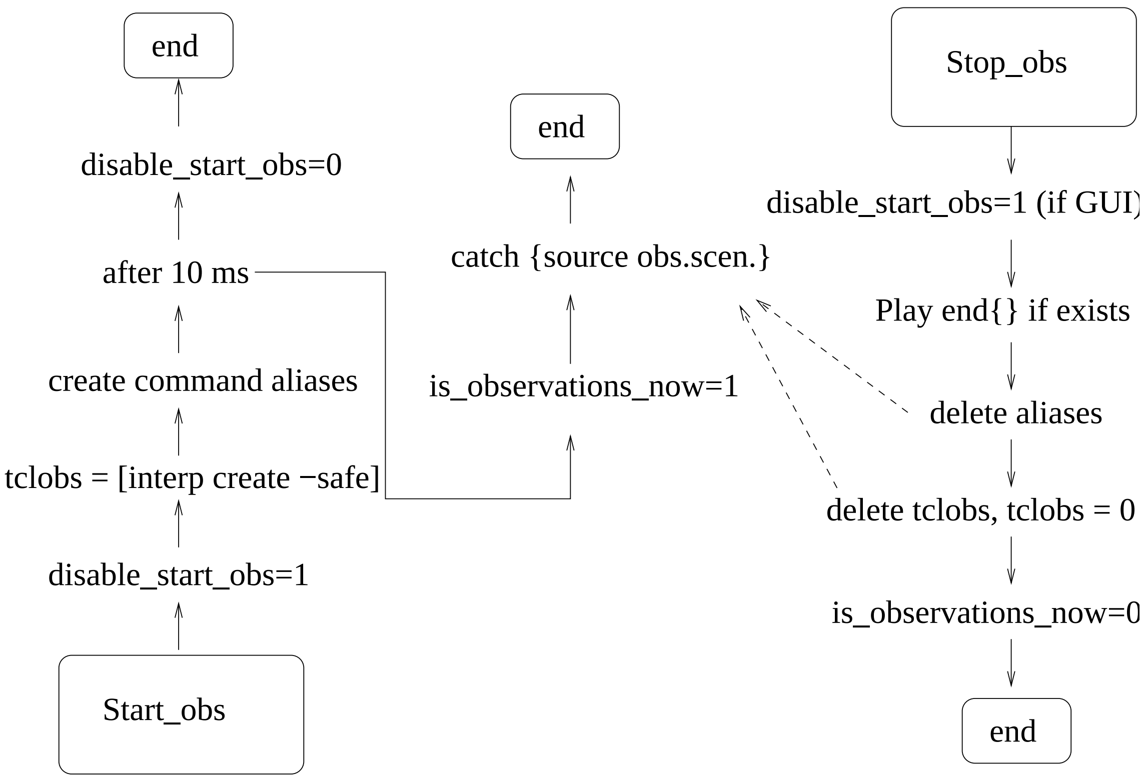


Figure 7: Control variables in starting and stopping the scenaria. Dashed arrows denote errors (emerging in scenaria during use of deleted aliased commands or attempt to call a command in deleted interpreter) which cause catch-construct to work and thus stop scenario evaluation.

When starting observations from the circumstance scenario, the is_observations_now{} function is used. It returns zero if the obs. scenario is not executed.

The switching of the control variables described above is sketched in Fig. 7 on the example of starting/stopping observations. Actions performed both in start_obs{}, stop_obs{} and start_scen{}, stop_scen{} are presented. Checking listed in "Conditions" to respective procedures above are not shown. In brief, the values of `tclobs` and `tclmon` are assigned to interp-initializers from the very beginning and indicate that start_*{} procedures are under way. When the process is finished and the respective script commands started execution, the final flag is set – is_observations_now-¿1 in start_obs{}. Start_monitor{} does not need such a final flag, since no such strong restrictions on its start apply.

While stopping the scenaria, the scenario procedure end{} is called first, if provided. As any other, such a call is done within the catch-construction, to protect SV against the errors in untrusted scenario scripts. Then, the respective interpreter is deleted. Note, that the delayed execution may cause an attempt to execute the next scenario command on a background in an interpreter which does not exist any more. This error is specially caught by the catch-construction in after-scripts started from start_scen{} and has a special errorCode IDELETE in Tcl which helps to differ it from other errors which may emerge from (erroneous and buggy) scenaria scripts. Such an error is a normal consequence of complicated time-diagrams of scenaria involving after-commands and is ignored when caught.

The signal that stopping of the scenario and, possibly, parking of involved components is over is assignment of the interpreter variable (tclobs or tclmon) to zero. This indicates to other parts of SV the potential to start the scenario again.

E.5 Possible errors/failures occurring in Supervisor

```
Startup errors:
ErrName Proc Error  error type

ENOCFG read_cfg no cfg sv.cfg found cfg
EBADCFG read_cfg Single word in cfg FILE, not a comment  cfg
ENOPCFG read_cfg No obligatory parameter PAR found in CFG cfg
ENOCMP connect Cannot connect to COMP connect
ENMCMP connect COMP ident IDENT does not match IDENT from cfg cfg
EBADSCE check_scen Scenario file not readable cfg



Execution errors:
ECMDDSC cmd  Attempt to send a command to disconnected COMP connection
ECMPNEX cmd  A command to non-existent component COMP cfg
ECMDLOS cmd Timeout waiting for COMP reply comp.fatal
ECMPDSC get_reply Lost connection to COMP connection
ECMDID get_reply Unexpected reply: no command COMID sent to COMP comp.prog
ECMDPAR get_reply Parsing reply: no param. value in REPLY comp.prog
ECMDLOW get_reply Timeout waiting for COMP reply (after wait) comp.fatal
ECMPFAT get_reply   ERROR reply: fatal error in COMP                comp.fatal
ECMPSTA get_reply ERROR reply: cmd cannot be executed: STATUS scen
ECMDKWD get_reply Unknown keyword: REPLY from COMP comp.prog
ENOPAR <comp>Access to non-existent component parameter cfg
ECMPNST stop_park COMP is busy after STOP NOW comp.prog
ECMDSCE start_scen   <error message from scenario evaluation>    scen
```

Actions due to errors:

Startup errors: exit with error message (the operator controls troubleshooting and SV restart)

Execution errors: occur after everything started up correctly

ECMDDSC: ignore without messages (optional comp. may be closed due to problems)

ECMDID, ECMPNST, ECMDPAR, ECMDKWD (dangerous: unpredictable software behavior): so far - behavior the same as for ECMDLOS etc.......

ECMDLOS, ECMDLOW, ECMPDSC, ECMPFAT: if some scenario procedure `error_handler` {`code comp`} is found – execute it. If returned value is 1 or (in interactive==1 mode) user selects to skip this error – return. Othervise: disconnect comp, issue error message (`emergency_sys`), if comp(optional) exists (whatever value!) - return; else – `stop_obs, stop_park, stop_monitor`, wait for `SV(revive_time)` (if positive) and restart SV script.

ECMPSTA: if status is BUSY - retry after status changes; if ERANG/ERSYN – ignore in case error is component-dialog-caused or treate as ECMPFAT; otherwise – ignore with error message

Unknown error code: causes error in SV and call of `emergency_sys`.

Error handling Error handling strategy is based on usage of the common error handler which may either return or exit depending on situation and after making some trouble-shooting actions. This handling of errors is concentrated in a single procedure `problem` {`message code [comp]` `[cmd]`} in namespace sv, which is called in case some error state occurs. The call is made with following parameters:

message error message (human-readable)

code error code to classify an error and select actions to perform in each particular case

comp (optional): component with which an error is associated

cmd (optional): command sent to the component which caused an error state

One or both last two parameters are empty defaults for errors which do not deal directly with components.

`problem`{} has a switch-body where, for groups of errors listed above, the actions described there are performed. In case the normal functioning of SV should be canceled, the `terminate`{} procedure is called and then SV exits. In some cases, after `SV(revive_time)` seconds, SV will restart again. Note, that this should be used with caution since the bug-caused restarts may block the system filling it up with non-terminating SV instances.

Before starting the switch-body described above, the checking is done whether SV is currently terminating its work. This is done to avoid the infinite recursion of problem{} procedure due to secondary errors occuring while terminating. The only exception is the loss of connection: the lost component is immediately disconnected to stop the events flow from its channel caused by the EOF state.

Since introduction of the (optional) `interactive` parameter, SV behaves here in two ways: if *interactive* – the choice for an operator is given to by-pass the error situation somehow, i.e. not simply terminate, which is the only action made upon the serious error in `interactive==0` mode.