The Multi-Run Simulation Environment

SimEnv

User's Guide for Version V1.03

by M. Flechsig, U. Böhm, Th. Nocke & C. Rachimow

Disclaimer of Warranty

We make no warranties, expressed or implied, that the programs and data contained in the software package and the formulas given in this document are free of error, or are consistent with any particular standard of merchantability, or that they will meet your requirements for any particular application. They should not be relied for solving a problem whose incorrect solution could result in injury to a person or loss of property. If you do use the programs or data or formulas in such a manner, it is on your own risk. We disclaim all liability for direct or consequential damages from your use of the programs and data.

The Multi-Run Simulation Environment SimEnv

User's Guide for Version V1.03 (26-Mar-2004)

by

Michael Flechsig	Potsdam Institute for Climate Impact Research Department Data & Computation, flechsig@pik-potsdam.de
Uwe Böhm	University of Potsdam Institute for Physics, boehm@pik-potsdam.de
Thomas Nocke	University of Rostock Institute of Computer Graphics, nocke@informatik.uni-rostock.de
Claus Rachimow	Potsdam Institute for Climate Impact Research Department Data & Computation, rachimow@pik-potsdam.de

SimEnv in the Internet:
http://www.pik-potsdam.de/topik/pikuliar/simenv/home

Potsdam Institute for Climate Impact Research
Telegrafenberg
14473 Potsdam, Germany
Phone ++49 – 331 – 288 2604
Fax ++49 – 331 – 288 2600
WWW http://www.pik-potsdam.de

University of Potsdam
Institute for Physics
Am Neuen Palais 10
14469 Potsdam, Germany
WWW http://physik.uni-potsdam.de

University of Rostock
Institute of Computer Graphics
Albert-Einstein-Str. 21
18059 Rostock, Germany
WWW http://wwwicg.informatik.uni-rostock.de

Contents

Tables

Figures

Examples

Executive Summary

SimEnv is a multi-run simulation environment that focuses on model evaluation and usage mainly for quality assurance matters and scenario analyses using sampling techniques. Interfacing models to the simulation environment is supported for a number of programming languages by minimal source code modifications and in general at the shell script level. Pre-defined experiment types are the backbone of SimEnv, enabling experimenting with numerical parameter, initial value, or driving forces adjustments of the model. The resulting multi-run experiment can be performed sequentially or in parallel. Interactive experiment post-processing makes use of built-in operator definitions, optionally supplemented by user-defined operators and applies operator chains on model output and reference data. Result output functions generated during post-processing can be evaluated with advanced visualization techniques within SimEnv.

Simulation is one of the cornerstones for research in Global Change. The aim of the SimEnv project is to develop a toolbox oriented simulation environment that enables the modeller to handle model related quality assurance matters (Saltelli *et al.*, 2000) and scenario analyses. Both research foci require complex simulation experiments for model inspection, validation and control design without changing the model in general.

SimEnv aims at model evaluation by performing simulation runs with a model in a co-ordinated manner and running the model several times. Co-ordination is achieved by pre-defined experiment types representing multi-run simulations.
According to the strategy of a selected experiment type for a set of so-called targets **t** which represent drivers, parameters, boundary and initial values of the model **M** a sample is generated before simulation and the targets are re-adjusted numerically before each single simulation run during the experiment. Each experiment results in a sequence of model outputs over the single runs for selected state variables **z** dependent on the target adjustments of the model. Model outputs can be processed and evaluated across the run ensemble specifically after simulation.

The following experiment types form the base of the SimEnv multi-run facility:

- Behavioural analysis
  Inspection of the model's behaviour in a space spanned from targets t with discrete numerical adjustments and a flexible inspection strategy for the whole space.
  For model verification, numerical validation, deterministic error analysis, deterministic control design, scenario analysis and spatial patch model applications.
- Monte-Carlo analysis
  Perturbations of targets t according to probability density functions. Determination of moments, confidence intervals and heuristic probability density functions for z in the course of post-processing.
  For error analysis, uncertainty analysis, verification and validation of deterministic models.
- Local sensitivity analysis
  Determination of model (state variables) sensitivity to targets p. Is performed by finite difference derivative approximations from M.
  For numerical validation purposes, model analysis, sub-model sensitivity.
- Optimization (in preparation)
  Iterative determination of optimal targets t for mono- or multi-criterial cost functions derived from z by gradient-free methods.
  For model validation (system - model comparison), control design, decision making.

SimEnv makes use of modern IT concepts. Model preparation for interfacing them to SimEnv is based on minimal source code manipulations by implementing function calls into Fortran-, C/C++-, Python- or GAMS-model source code for p-adjustments and model output. Additionally, an interface at shell script level is available.
In experiment preparation an experiment type is selected and equipped numerically. Experiment performance supports local, remote, and parallel architectures.
Experiment-specific model output post-processing enables navigation in the experiment - model output space and interactive filtering of model output and reference data by application of built-in and user-defined post-processing operator chains.

Result evaluation is dominated by application of pre-formed visualization modules.
SimEnv model output as well as model output post-processing offer data interfaces for NetCDF, IEEE compliant binary and ASCII format for a more detailed post-processing outside SimEnv.

SimEnv key features:

- Support of key working techniques in experimenting with models:
  SimEnv enables model evaluation, uncertainty and scenario analyses in a structured, methodologically sound and pre-formed manner applying sampling techniques.
- Run ensembles instead of single model runs:
  Model evaluation by multi-run simulation experiments
- Availability of pre-defined multi-run simulation experiment types:
  To perform an experiment only the targets to experiment with (parameters, drivers, initial values, ...) and rules how to re-adjust them numerically have to be specified.
- Simple model interface to the simulation environment:
  There are model coupling functions mainly to re-adjust an experiment target and to output model results for later post-processing. Model coupling and finally communication between the model and SimEnv can be done at the model language level by incorporating coupling functions into model source code (C, Fortran and Python) or can be done at the shell script level within shell-scripts. Additionally, there is a special interface for GAMS models.
- Support of distributed models:
  Independently on the kind distributed models are coupled they can be interfaced to SimEnv.
- Parallelization of the experiment:
  This is a prerequisite for a lot of simulation tasks.
- Operator-based experiment post-processing:
  Chains of built-in and user-defined operators enable interactive experiment post-processing based on experiment model output and reference data including general purpose and experiment-specific operators.
- Graphical experiment evaluation:
  For post-processed model output
- Support of standard data formats:
  Output from the model as well from the post-processor can be stored in NetCDF or IEEE compliant binary format.

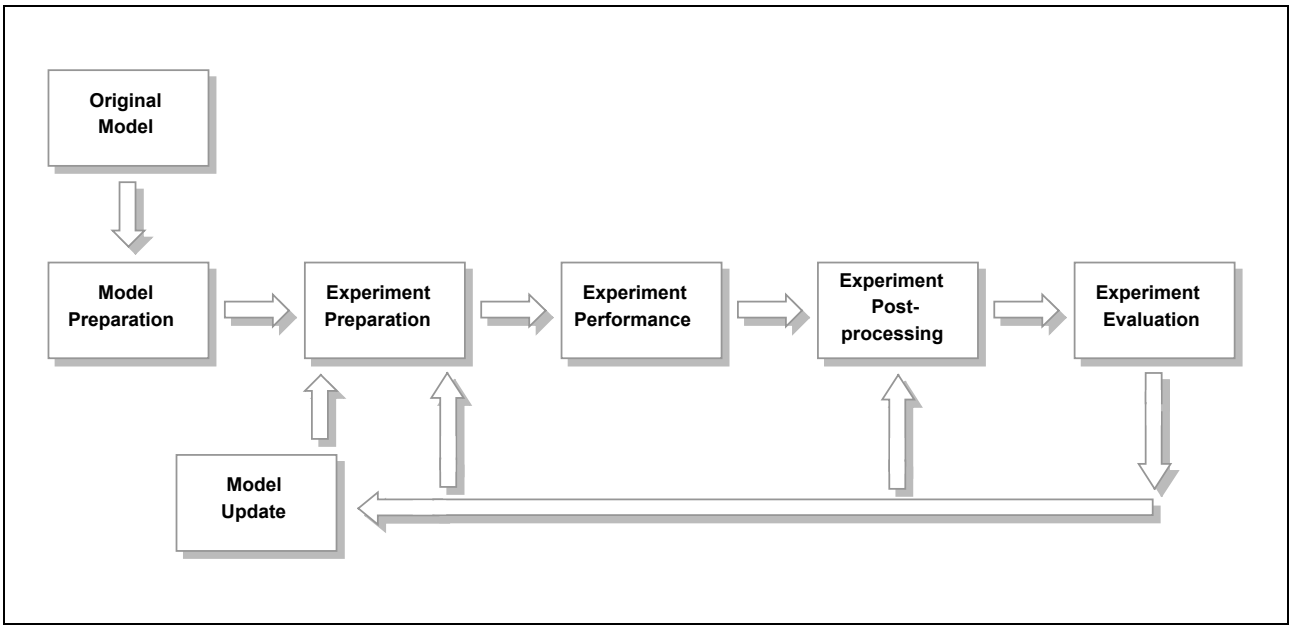


Fig. 0.1 *SimEnv system design*

1 About this Document

In this chapter document conventions are explained. Within the whole document one reference example model is used to explain application of SimEnv. Examples are always located in grey boxes.

1.1 Document Conventions

Character / string	Meaning
< ... >	angle brackets enclose a placeholder for a string
{ ... }	braces enclose an optional element
[... | ... | ...]	square brackets enclose a list of choices, separated by a vertical bar
nil	stands for the empty string (nothing)
`monospace`	indicates SimEnv example code

Tab. 1.1 *Document conventions*

Tab. 1.2 summarizes the main placeholders used in this document.

Placeholder	Description
<file_name>	name of a data file
<GAMS_model>	name of a GAMS model
<model>	model name to start a SimEnv service with
<nil>	the empty string
<path>	path to a file
<res>	integer post-processor output file number 1, 2, ..., 99
<res_char>	character post-processor output file number 01, 02, ..., 99
<run>	integer single run number 0, 1, ... within an experiment
<run_char>	character single run number 000000, 000001, ... within an experiment
<sep>	sequence of white spaces as item separators in user-defined files
<string>	any string
<target_def_val>	default value of a target according to <model>.edf
<target_name>	name of a target to experiment with
<value_list>	list of values in explicit or implicit notation according to Tab. 11.4

Tab. 1.2 *Placeholders in this document*

1.2 Examples

Examples in this document refer to a hypothetical global simulation model. It is to describe dynamics of atmosphere and biosphere at the global scale over 200 years. Lateral (latitudinal and longitudinal) model resolution differs for different model implementations (see below), temporal resolution is at decadal time steps. Additionally, atmosphere is structured vertically into levels.
Models with name world_* are assumed to map lateral (latitudinal/longitudinal) and vertical (level) fluxes and demanding that's why for computing state variables for the whole globe.
In the model pixel_f state variables are calculated for one pixel (for one single latitude - longitude constellation).

Model state variable	Description	Defined on	Data type
atmo	aggregated atmospheric state	lat x lon x level x time	float
bios	aggregated biospheric state at land masses (defined between 83°N and 60°S latitude at land masses, i. e., without Antarctic)	lat x lon x time	float
atmo_g (not for model pixel_f)	aggregated global state derived from atmo for level 1	time	int
bios_g (not for model pixel_f)	aggregated global state derived from bios	-	int

Dynamics of all of these model variables depend on model parameters p1, p2, p3 and p4.

With this SimEnv release the following model versions are distributed:

Model	Model interface example for	Resolution		
		lateral lat x lon	vertical # of levels	temporal # of time steps
world_f	Fortran	4 x 4	4: 1, 7, 11, 16	20
world_c	C	4 x 4	4: 1, 7, 11, 16	20
world_cpp	C++	4 x 4	4: 1, 7, 11, 16	20
world_py	Python	4 x 4	4: 1, 7, 11, 16	20
world_sh	script level	4 x 4	4: 1, 7, 11, 16	20
pixel_f	Fortran	without, implicitly by experiment as 4 x 4	4: 1, 7, 11, 16	20
world_f_1x1	Fortran	1 x 1	16: 1 - 16	20

The only example that does not refer to the above model type is that for GAMS model interface to SimEnv (chapter 5.4 at page 24).

Examples are generally placed in grey boxes.

Examples that are available in the corresponding examples directory of $SE_HOME are marked as such in the lower right corner of an example box.

Example 1.1 *Example layout*

2 Getting Started

In this chapter a quick start tour is described. Without going into details the user can get an impression how to apply SimEnv and which user files are essential to use the simulation environment.

- SimEnv is implemented under AIX at IBM's RS6000.
- Set the operating system environment variable **SE_HOME** to /usr/local/simenv/bin in your .profile file and export it for the Korn shell.
- For interfacing Python and GAMS models to SimEnv extend your operating system environment variable **PYTHONPATH** by $SE_HOME, include it in your .profile file and export it for the Korn shell.
- Change to a working directory you have full access rights.
- Get basic information on SimEnv by entering
  $SE_HOME/simenv.hlp
  at the operating system prompt.
- Select a model implementation language <lng> you want to check SimEnv with the model from Example 1.1 at page 4:

<lng> =		
	f	for Fortran
	c	for C
	cpp	for C++
	py	for Python
	sh	for shell script level

  For the test model contents check Example 1.1 at page 4. For a GAMS model example check chapter 5.6 at page 25.
- Start from the working directory the shell script
  $SE_HOME/simenv.cpy world_<lng>
  to copy model world_<lng> model and experiment related files to this working directory.
- Copy the file world.edf_c to world_<lng>.edf
- Check for
  - The SimEnv configuration file **world_<lng>.cfg** general configurations of SimEnv
  - The model output description file **world_<lng>.mdf** available model variables
  - The model **world_<lng>.<lng>** implementation of the model
  - The model shell script **world_<lng>.run** wrapping the model executable
  - The experiment description file **world_<lng>.edf** experiment definition
  - The post-processing input file **world.post_c** post-processor expression sequence
  - The macro description file **world_<lng>.mac** macros for the post-processor
  - The operator description file **world_<lng>.opr** description of user-defined operators
  - The user-defined operators **usr_opr_<opr>.f** code of user-defined operator <opr>
- Start a complete SimEnv session by
  $SE_HOME/simenv.cpl world_<lng> -1 world.post_c
  - SimEnv files will be checked
  - The experiment will be prepared
  - The experiment will be performed machine (select the login machine on request)
  - Model output post-processing will be started for this experiment
    - With the post-processing input file world_post_c and following
    - Interactively: Enter any expression and finish post-processing by entering a single <return>
  - Visualization of post-processed results will be started **(*)**

  or
  - Start **$SE_HOME/simenv.chk** world_<lng>
    to check model and experiment files.
  - Start **$SE_HOME/simenv.run** world_<lng>
    to prepare and perform a simulation experiment.
  - Start **$SE_HOME/simenv.rst** world_<lng>
    to restart a simulation experiment.
  - Start **$SE_HOME/simenv.res** world_<lng> {[new | append | replace]} {<run>}

to post-process the last simulation experiment over the whole run ensemble or for run number <run> and to create a new / append to / replace the result file <model>.res<res_char>.[nc | ieee | ascii] with the highest two-digit number <res_char>. <res_char> (can range from 01 to 99).

- Start **$SE_HOME/simenv.vis** world_<lng> {[latest | <res_char>]} **(*)**
  to visualize output from the latest post-processing output file world_<lng>.res<res_char>.nc or that with number <res_char> with the highest two-digit number <res_char>. <res_char> can range from 01 to 99.

- Check in the working directory the model interface and experiment performance log-files
  world_<lng>.mlog and world_<lng>.elog.
- Start **$SE_HOME/simenv.dmp** world_<lng> | more
  to dump a SimEnv model or post-processor output file.
- Start **$SE_HOME/simenv.cln** world_<lng>
  to wrap up a simulation experiment.
- Get the usage of all commands by entering a command without arguments.
- To run other simulation experiments and/or output in other data formats modify
  - world_<lng>.cfg
  - world_<lng>.edf
  - world_<lng>.mdf
  - world_<lng>.<lng> and/or
  - world_<lng>.run
- To experiment with other models replace world_<lng> by <model> as a placeholder for the name of any other model.

(*): To get access rights for the visualization server check the SimEnv service
$SE_HOME/simenv.key <user_name>
in chapter 10.2 at page 83.

3 Version 1.03

This chapter summarizes differences between the current and the previous SimEnv release, limitations, and bugs and workarounds.

3.1 What is New?

Type	Check / see	At page	Description
new	Tab. 5.6	30	New sub-keyword options for GAMS description file <model>.gdf
new	Chapter 12.3 Example 15.8	98 122	Changes in structure of result output file <model>.res<res_char>.[ieee | ascii] This file is now complemented by a structure description file <model>.inf<res_char>.[ieee | ascii]
new	Example 15.3	115	Description of model interface for C++ models
			Bug fixes

Tab. 3.1 *SimEnv changes in version 1.03*

Upgrade type	Upgrade action
mandatory	Re-link all models
mandatory	Update yourt private SimEnv result output import interface for IEEE and ASCII output

Tab. 3.2 *User actions to upgrade to version 1.03*

3.2 Known Bugs and Their Workarounds

Where	Experiment post-processing: Behavioural analysis / result output to NetCDF
Bug	When applying operator behav non-monotonic target adjustments are transferred to NetCDF output in a wrong manner.
Workaround	Specify only monotonic target adjustments in <model>.edf

Tab. 3.3 *Known bugs and their workarounds*

3.3 Limitations

- Only accessible under AIX
- Without experiment specific operators for local sensitivity analysis in experiment post-processing:
  Only a selected single run can be post-processed for this experiment type.
- No C-interface to write user-defined operators
- Preliminary graphical evaluation of post-processed model output
- Graphical user interface only for graphical evaluation

4 Experiment Types

SimEnv supplies a set of pre-defined multi-run experiment types. Each experiment type addresses a special experiment class for performing a simulation model several times in a co-ordinated manner. In this chapter an overview on the available experiment types is given from the viewpoint of system's theory.

4.1 General Approach

SimEnv supplies a set of pre-defined multi-run experiment types, where each type addresses a special multi-run experiment class for performing a simulation model or any algorithm with an input - output transition behaviour.
In the following, the general SimEnv approach will be described for time dynamic simulation models, because this class forms the majority of SimEnv applications. All information can be transformed easily to any other algorithm.

Based on systems' theory, each time dynamic model M can be formulated - without limitation of generality - for the time dependent, time discrete, and state deterministic case as

$$M: \quad Z(t) = ST (Z(t-\Delta t) ,..., Z(t-n*\Delta t) , P , X(t) , Z_0 , B)$$

with

ST	state transition description
Z	state variables' vector
P	parameter vector
X	input (driving forces) vector
Z_0	initial value vector
B	boundary value vector
t	time
Δt	time increment
n	time delay

The output vector Y is a function of the state vector Z, parameters P, drivers X, and initial values Z_0:

$$Y(t) = OU (Z(t) , P , X(t) , Z_0).$$

Model behaviour Z is determined for fixed n and Δt by state transition description ST, parameters P, driving forces X, initial values Z_0, and boundary values B. Manipulating and exploring model behaviour in any sense means changing these four model components. While state transition description ST reflects mainly model structure and is quite complex to change, each component of the driving forces vector X normally is a time-dependent vector.

Introduction of additional technical parameters P_{tech} can reduce the complexity of handling a model with respect to the five model components, described above: Changes in state transition description ST can be predetermined in the model by assigning values of a technical parameter p_{tech} to alternative submodel versions, which are switched on or off by these values. Additionally, each component of the driving forces vector X can be combined with technical parameters in different ways:

- By selecting special driving forces dependent on the technical value
- By manipulating the driving forces with the parameter value (e.g., as an additive or multiplicative adjustment)
- By parametrizing the shape of a driving force

When this has been done, the model behaviour finally depends only on the parameters P, the initial values Z_0, and the boundary values B. From the methodical point of view there is no difference between parameters, initial values and boundary values, because all are considered as constant during one model run. That

is why in the following the term **target** stands as a placeholder for all the four model components parameters, drivers, initial values and boundary values. All targets form the target set T:

$$T = \{ P, X , Z_0 , B \}$$

and

$$Z = ST(T).$$

In the following,

$$T_m = (t_1 ,..., t_m) \qquad m > 0$$

stands for a subset of the target set T that spans up an m-dimensional sub-space of T by selected model targets $(t_1 ,..., t_m)$ from T
and

$$T_{mn} = \begin{pmatrix} t_{11} & ... & t_{1m} \\ ... & & ... \\ t_{n1} & ... & t_{nm} \end{pmatrix} \qquad m > 0,\ n > 1$$

stands for a numerical sample for T_m of size n and finally for m*n values representing in any sense T_m.
In the set of all T_{mi} (i > 1) one extraordinary sample T_{m1} exists that matches the nominal (default) numerical target constellation for the model M.
If $\{ \}_n$ denotes the dynamics of the model M over a sample of size n then it yields:

$$\{ Z \}_n = \{ ST(T_{mn}) \}_n .$$

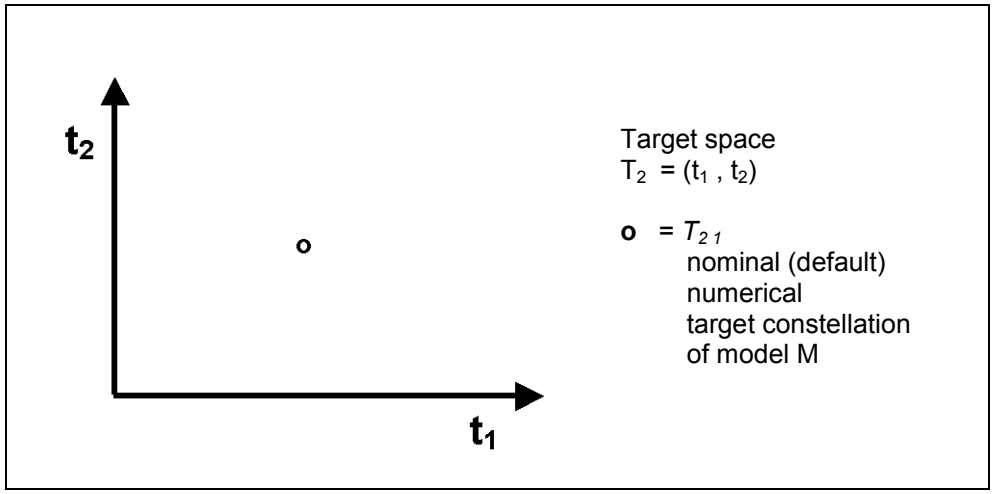


Fig. 4.1 *Target space*

SimEnv supports different sampling stategies and performance of multi-run experiments where m targets are readjusted numerically for each of n single simulation runs. Central goal is to study depencency of the model dynamics on target adjustments. For simulation purposes in SimEnv experimentation with the model M over T_{mn} is based on the assumption that dynamics of M for each representative from the sample is indepent from all other representatives, which is fulfilled in general. This results in the possibility to form a run ensemble for performing the model M with n single model runs from the sample T_{mn}.

SimEnv experiment types differ in the way T_m is sampled to get T_{mn}. There are deterministic and non-deterministic sampling strategies that offer a broad range of techniques for

- Experimentation with models
- Post-processing model output results
- Interpreting results with respect to uncertainty and sensitivity matters of models.

The experiment types are described in detail in the following.

4.2 Behavioural Analysis

Behavioural analysis uses a deterministic strategy to sample T_m. It is the inspection of the model in the target space T_m where inspection points are set in a regular and well structured manner.

Behavioural analysis can be interpreted and used in different ways:

- For scenario analysis:
  to show how model behaviour changes with changes of target values
- For numerical validation purposes:
  to determine target values in such a way that the output vector matches with measurement results of the real system
- For deterministic error analysis:
  to analyse how the model error is dependent on target errors
- For a simulation-based control design:
  to determine target values in such a way that a goal function becomes an extreme

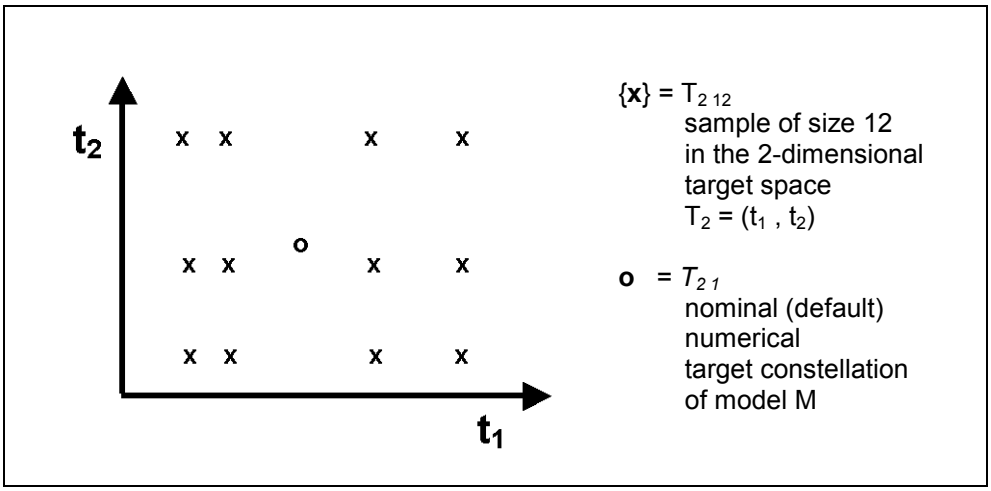


Fig. 4.2 *Sample for a behavioural analysis*

SimEnv behavioural analysis sampling strategy is a generalization of the one-dimensional case for T_1, where the model behaviour is scanned in dependence on deterministic adjustments of one target t_1. The general case for T_m demands a strategy for scanning m-dimensional spaces in a flexible manner. Based on the predecessors of SimEnv (Wenzel *et al.*, 1990, Wenzel *et al.*, 1995, Flechsig, 1998) subspaces of the m-dimensional target space can be scanned on the subspace diagonal (parallel in a one-dimensional hyper-space) or completely for all dimensions (combinatorially on a grid) and both techniques can be combined. Besides this regular scanning method an irregular technique is possible.

The resulting number of single simulation runs for the experiment depends on the number of target samples per dimension of the scanned target space and from the selected scanning method. An experiment is described by the names of the involved targets, their numerical adjustments and their combination (scanning method). Model output post-processing resolves the scanning method again and outputs results as projections on multi-dimensional target subspaces.

Fig. 4.3 describes the regular scanning technique by an example. In the left scheme (a) the two-dimensional target space $T_2 = (p_1 , p_2)$ is scanned combinatorially, resulting in 4*4 = 16 model runs, while the middle scheme (b) represents a parallel scanning of these two targets at the diagonal by 1+1+1+1 = 4 model runs. The scheme (c) at the right side shows a complex scanning strategy of the 3-dimensional target space $T_3 = (p_1 , p_2 , p_3)$ with (1+1+1+1)*3 = 12 model runs. Each filled dot represents a single model run.

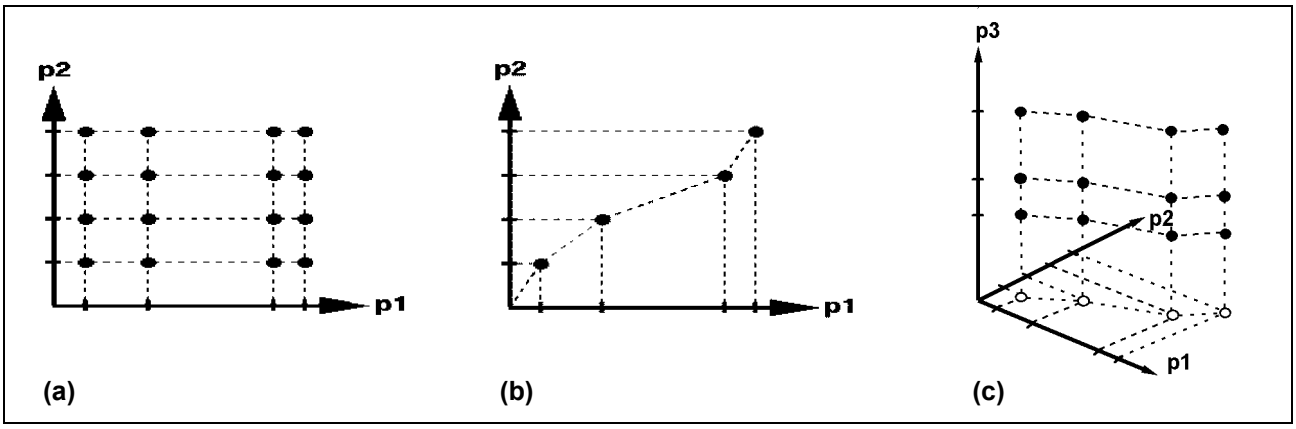


Fig. 4.3 *Behavioural analysis: Scanning multi-dimensional target spaces*

4.3 Monte-Carlo Analysis

Monte-Carlo analysis uses a non-deterministic strategy to sample T_{mn}. A Monte-Carlo experiment in SimEnv is a perturbation analysis with pre-single run target perturbations.

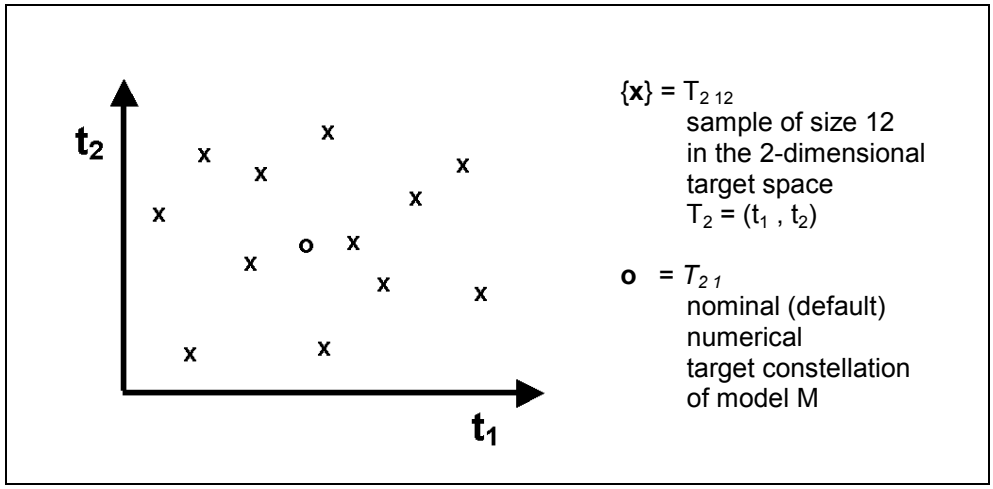


Fig. 4.4 *Sample for a Monte-Carlo analysis*

Theoretically, with a Monte-Carlo analysis moments of a state variable z can be computed as

$$M^{(k)}\{z\} = \int_{T_m}^{\cdots} z(T_m)^k * pdf(T_m)\, dT_m$$

with	$M^{(k)}\{z\}$	k-th moment of the state variable z with respect to the probability density function pdf
	$z(T_m)$	state variable z as a function of T_m
	$pdf(T_m)$	probability density function of T_m

By interpreting the probability density function $pdf(T_m)$ as the error distribution in the target space T_m it is possible to study error propagation in the model. On the other hand Monte-Carlo analysis can be interpreted as a stochastic error analysis, if there are measurements of the real system for z.

For a numerical experiment in SimEnv it is assumed that the probability density function $pdf(T_m)$ can be decomposed into independent probability density functions pdf_i for all targets t_i of T_m:

$$pdf(T_m) = \prod_{i=1}^{m} pdf_i(t_i)$$

and the m-dimensional integral is approximated by a sequence of n single simulation runs of the model where the numerical target values t_{ij} of t_i ($1 \le i \le m$, $1 \le j \le n$) are sampled according to the probability density function pdf_i.
On the basis of these assumptions, the statistical measures in Tab. 4.1 can be computed during performance of a post-processing session from a Monte-Carlo analysis with n simulation runs resulting in n realizations z_1 ,..., z_n of the state variable z:

Statistical measure	Definition (*)
minimum	$min(z) = \min(z_i)$
maximum	$max(z) = \max(z_i)$
sum	$sum(z) = \sum z_i$
average	$M^{(1)}(z) = \sum z_i / n$
variance	$M^{(2)}(z) = \sum (z_i - z^{(1)})^2 / (n - 1)$
skewness	$M^{(3)}(z) = \sum (z_i - z^{(1)})^3 / n * (\sum (z_i - z^{(1)})^2 / (n - 1))^{3/2}$
kurtosis	$M^{(4)}(z) = (\sum (z_i - z^{(1)})^4 / n * (\sum (z_i - z^{(1)})^2 / (n - 1))^2) - 3$
range	$rng(z) = max(z) - min(z)$
geometric average	$avgg(z) = (\prod z_i)^{1/n}$
harmonic average	$agvh(z) = n / \sum(1 / z_i)$
weighted average	$avgw(z) = \sum z_i * w_i / \sum w_i$ w : weight
correlation	$cor(z1,z2) = \sum (z1_i - z1^{(1)}) * (z2_i - z2^{(1)}) / \sqrt{ \sum (z1_i - z1^{(1)})^2 * \sum (z2_i - z2^{(1)})^2 }$
covariance	$cov(z1,z2) = \sum (z1_i - z1^{(1)}) * (z2_i - z2^{(1)}) / (n - 1)$
linear regression coefficient	$reg(z1,z2) = \sum (z1_i - z1^{(1)}) * (z2_i - z2^{(1)}) / \sum (z1_i - z1^{(1)})^2$
median	$med(z)$ = middle value from increasingly ordered $\{ z_i \}$ (n = odd); mean of the two middle values from $\{ z_i \}$ (n = even)
quantile	$qnt^{(p)}(z)$ = that value from increasingly ordered $\{ z_i \}$ which corresponds to a cumulative frequency of n*p; $qnt^{(0.5)}(z) = med(z)$
confidence interval boundaries	$cnf^{(\alpha)}(z) = z^{(1)} \pm t_{\alpha,n-1} \sqrt{z^{(2)} / n}$ with level of error α = 0.1%, 1% and 5%; $t_{\alpha,n}$: significance boundaries of Student distribution
heuristic probability density function	$hgr^{(class)}(z)$ = number of z_i with $class_{min} \le z_i < class_{max}$

Tab. 4.1 *Statistical measures*

(*): indices for sums $\sum$, products $\prod$ and extremes run from 1 to n: $\sum_{i=1}^{n}$ $\prod_{i=1}^{n}$ $\min_{i=1,...,n}$ $\max_{i=1,...,n}$

Tab. 4.2 summarizes these probability density functions (Bohr, 1998) that are pre-defined in SimEnv for targets to be perturbed. Additionally, SimEnv offers to import random number samples in the course of experiment preparation.

Distribution	Short-cut	Probability density function pdf		Distribution parameters	
uniform	U(a,b)	pdf(x) = $\frac{1}{b-a}$	if x $\in$ [a,b]	a b	lower boundary upper boundary > a
		pdf(x) = 0	otherwise	it is:	mean = (a+b) / 2 standard deviation = $\sqrt{(b\text{-}a)^2 / 12}$
normal	N(μ,σ^2)	pdf(x) = $\frac{1}{\sigma\sqrt{2\pi}}\exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$		μ σ	mean standard deviation > 0
lognormal	L(μ,σ^2)	pdf(x) = $\frac{1}{x\sigma\sqrt{2\pi}}\exp\left(-\frac{(\ln x-\mu)^2}{2\sigma^2}\right)$	if x > 0	μ σ	 > 0
		pdf(x) = 0	otherwise	it is:	ln(x) ~ N(μ,σ^2)
exponential	E(μ)	pdf(x) = $\frac{1}{\mu}\exp\left(-\frac{x}{\mu}\right)$	if x > 0	μ	mean > 0
		pdf(x) = 0	otherwise	it is:	standard deviation = μ

Tab. 4.2 *Probability density functions*

The number of runs to be performed during a Monte-Carlo analysis has to be specified. An experiment is described by the targets involved in the analysis, their distribution and the appropriate distribution parameters.

4.4 Local Sensitivity Analysis

Local sensitivity analysis uses a deterministic sampling stategy in ε-neighbourhoods of the numerical default constellation T_{m1} of the model M. For each target t_i from the nominal target constallation T_{m1} and each ε_j from the ε-neighbourhoods (ε_1 ,…, ε_k) two members (t_1 ,..., t_{i-1} , $t_i \pm \varepsilon_j$, t_{i+1} ,..., t_m) of the resulting sample are generated. The sample size n is given by 2*m*k. Running the model at this sampling set serves to determine sensitivity functions.

In classical systems' theory, model sensitivity of a model state variable z with respect to a target t is the partial derivative of z after t. In the numerical simulation of complex systems finite sensitivity functions are preferred, because they can be obtained without model enlargements or re-formulations. They are linear approximations of the classical model sensitivity measures (Wierzbicki, 1984).

Local sensitivity functions can be used for localizing modification-relevant model parts as well as control-sensitive targets in control problems. On the other hand, identification of robust parts of a model or even complete robust models makes it possible to run a model under internal or external disturbances. Sensitivity analysis in SimEnv post-processing is based on finite sensitivity functions, which are defined as in Tab. 4.3.

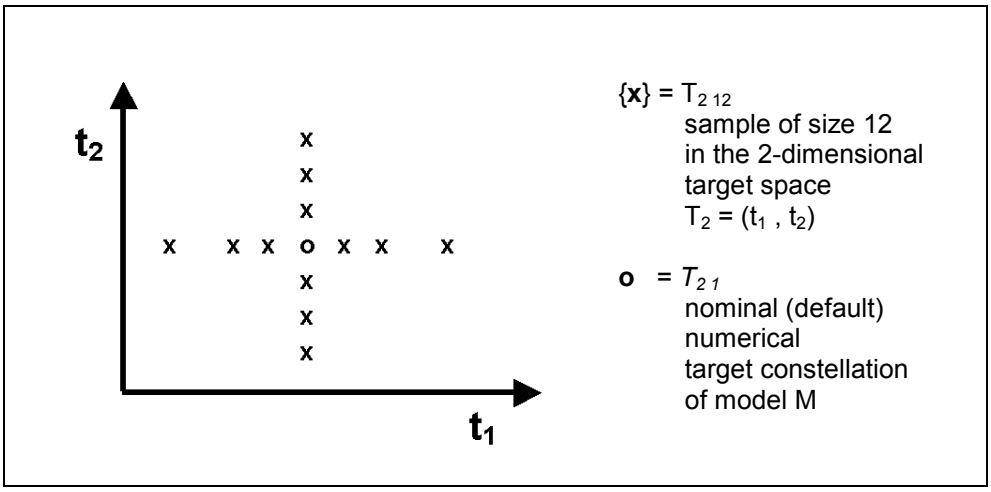


Fig. 4.5 *Sample for a sensitivity analysis*

Local sensitivity function	Definition
linear	$lin^{\pm}(z,\varepsilon) = \frac{z(t \pm \varepsilon) - z(t)}{\varepsilon}$
squared	$sqr^{\pm}(z,\varepsilon) = \frac{(z(t \pm \varepsilon) - z(t))^2}{\varepsilon}$
absolute	$abs^{\pm}(z,\varepsilon) = \frac{\lvert z(t \pm \varepsilon) - z(t) \rvert}{\varepsilon}$
relative 1	$rel1^{\pm}(z,\varepsilon) = \frac{z(t \pm \varepsilon) - z(t)}{z(t) * \varepsilon}$
relative 2	$rel2^{\pm}(z,\varepsilon) = \frac{\frac{z(t \pm \varepsilon) - z(t)}{z(t) * \varepsilon}}{\frac{\varepsilon}{t}}$
symmetry test	$sym(z,\varepsilon) = \frac{z(t + \varepsilon) - z(t - \varepsilon)}{\varepsilon}$

Tab. 4.3 *Local sensitivity functions*
for a selected target t from T_{m1} and a selected ε from (ε_1 ,…, ε_n)

Accordingly, local sensitivity of the model to a target is always expressed as the sensitivity of a model's state variable z, usually at a selected time step within a surrounding ε of a target value t. That is why the conclusions drawn from a local sensitivity analysis are only valid locally with respect to the whole target space. Additionally, local sensitivity functions only describe the influence of one target t_i from the whole vector T_m on the model's dynamics.

Linear, squared and absolute local sensitivity functions allow comparison of the influence of various targets on the same state variable. The relative local sensitivity functions are suited to comparing the sensitivity of the same target on different state variables, because of the normalization effect of the nominal state variable z(t) and the nominal target value *t*. The symmetry test will return zero if the state variable z shows a symmetrical behaviour in the surrounding of the nominal value of the target *t*.

A local sensitivity experiment is described by the names of the targets t to be involved and the increments ε. The number of runs for the experiment results from the number of targets and increments: two runs per tar-

get for each increment plus one run with the default values of the targets. Local sensitivity functions are calculated during model output post-processing.

5 Model Interface

To use any model within SimEnv it has to be coupled to the simulation environment. SimEnv offers easy coupling techniques at programming language and shell script level. While at language level SimEnv function calls have to be implemented into model source code to adjust experiment targets, i. e. model parameters, initial values or boundary values of the current single run out of the run ensemble numerically and to output simulation results, at the shell script level communication between the simulation environment and the model can be based on operating system information exchange methods. To plug the model into the simulation environment the variables of the model to be output during experiment performance and to be post-processed during model output processing have to be declared in the model output description file <model>.mdf. Additionally, the model itself has to be wrapped into a shell script <model>.run.
Model interfacing is related to transferring adjusted numerical values of model targets under investigation from the simulation environment to the model and to transferring model variables under investigation from the model to the simulation environment for later post-processing. Coupling is supported at the programming language level for C/C++, Fortran, Python, and GAMS programming languages, the model is implemented in and the shell script shell script level.

5.1 Coordinate and Grid Assignments to Variables

To each variable

- Dimensionality **dim(variable)**
- Extents **ext(variable,i)** with i=1,...,dim(variable)
- Coordinates **coord(variable,i)** with i=1,...,dim(variable)

are assigned to. The dimensionality is the number of dimensions, an extent is related to each dimension and represents the number of elements in that dimension. Extents are always greater than 1. To each dimension a coordinate is assigned to. Coordinates have a name and from all coordinate values the coordinate is defined for a subset is assigned to the extent of the dimension of the variable. Variables of dimensionality 0 do not have a coordinate assignment.
A variable of dimensionality n corresponds with an n-dimensional array, a variable of dimensionality 0 is a scalar.

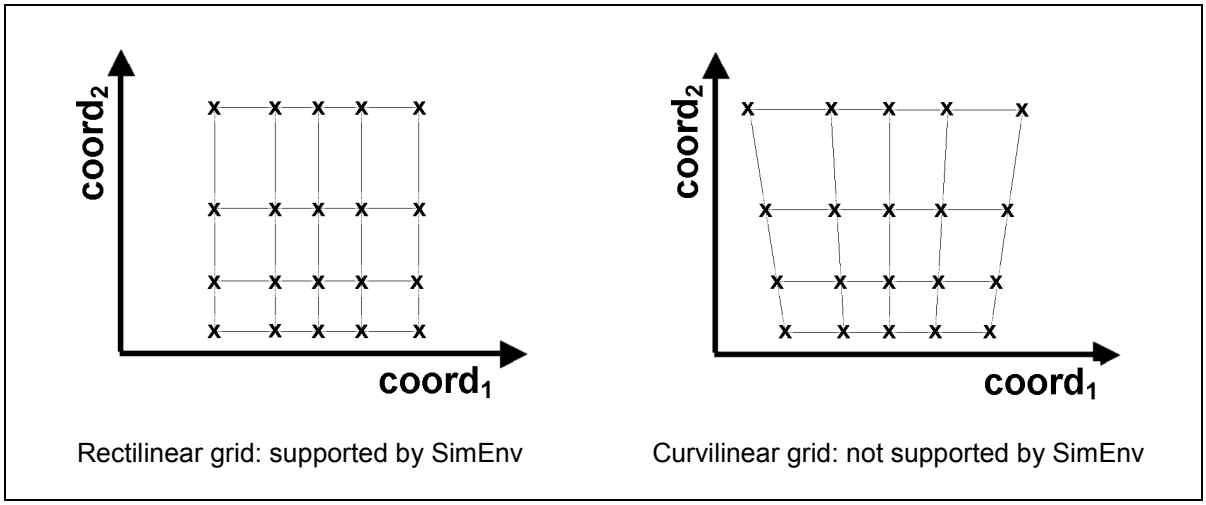


Fig. 5.1 *Grid types*

Additionally, coordinate axes are defined. Each coordinate axis a strictly monotonic sequence of coordinate values, a description and a unit is assigned to. For reasons of simplification in model output post-processing coordinate axes are assumed as curvilinear.
Each dimension of a variable with a dimensionality > 0 a complete coordinate axis or a part of a coordinate axis is assigned to. Consequently, each variable with a dimensionality > 0 is defined on a coordinate system formed from the assigned coordinates. For reasons of simplification in result evaluation with visualization techniques coordinate systems are assumed as rectilinear (orthogonal with variable distances between adjacent coordinate values). The model variable values then exist on the grid, spanned up from the coordinate values of the coordinate axes (see Fig. 5.1).

Since coordinate axes can be assigned to model variable dimensions in a flexible manner, model variables can exist on the same coordinate system or completely or partially disjoint coordinate systems.

5.2 Model Output Description File <model>.mdf

In the model output description file <model>.mdf the model variables are declared that are to be output by a SimEnv model coupling interface function in the model (code) and are to be post-processed after experiment performance. Additionally, coordinate axes are defined and flexibly assigned to model variables. Consequently, a model variable always is defined on a coordinate system, formed from the assigned coordinates to the variable.

<keyword>	**<name>**	**<sub-keyword>**	**Line type**	**Max. number of lines**	**<info>**	**Explanation**
mdf	<nil>	descr	o	any	<string>	model description
coordinate	coordinate_name (co_name)	descr	o	1	<string>	coordinate axis description
		unit	o	1	<string>	coordinate axis unit
		values	m	1	<value_list>	strictly monotonic sequence of coordinate values co_vals (for syntax see Tab. 11.4)
variable	variable_name	descr	o	1	<string>	variable description
		unit	o	1	<string>	variable unit
		type	m	1	see Tab. 5.2	variable type in the simulation model
		coords	c1	1	co_name_1 , ... , co_name_n	assigns a coordinate axis by its name to each dimension of the variable. Determines in this way implicitly the dimensionality n of the variable.
		coord_extents	c2	1	co_val_{11}:co_val_{12} , ... , co_val_{n1}:co_val_{n2}	assigns start and end coordinate value from each coordinate axis to the variable. If missing all coordinate values will be used from all assigned coordinates.
		var_extents	c1	1	vi_ext_{11}:vi_ext_{12} , ... , vi_ext_{n1}:vi_ext_{n2}	assigns start and end index for each dimension to the variable. Indices can be used to address the variable during post-processing.

Tab. 5.1 *Elements of a model output description file <model>.mdf*

Each model variable has a name, a dimensionality and assigned extents, a data type, a description and a unit. The name should correspond with the name of the variable in the simulation model code. Association between these two names is achieved by the SimEnv coupling function simenv_put_* (see below).

<model>.mdf is an ASCII file that holds this information. It follows the coding rules in chapter 11 at page 91 with the keywords, names, sub-keywords, and info as in Tab. 5.1.

To Tab. 5.1 the following additional rules and explanations apply:

- For the description of **line type** check Tab. 11.3 at page 92.
- Coordinate and variable names must differ from target names in experiment description (see chapter 0) and from built-in and user-defined operator names for model output post-processing (see chapter 8.6.2).
- Assignment of coordinate axes to variable dimensions and consequently of a grid to a variables is only valid for model output post-processing. Normally, the simulation model itself will also exploit the same grid structure. Nevertheless, the grid structures of the model are defined autonomously in the model in a explicit or implicit manner and do only correspond with the grid structure in the model output description file symbolically.
- Model variables with dimensionality 0 are not assigned to a coordinate axis.
- The values of a coordinate have to be ordered in a strictly monotonic sequence. They may be non-equidistant and may be ordered in a decreasing sequence.
- With the sub-keyword **coord_extents** only a portion of coordinate values of a coordinate axis can be assigned to a dimension of a variable. This portion is addressed by its begin and end value co_val_{i1} and/or co_val_{i2}. The number of coordinates of the portion has to be greater than 1.
  $co\_val_{i1} > co\_val_{i2}$ for strictly increasing values of coordinates
  $co\_val_{i1} < co\_val_{i2}$ for strictly decreasing values of coordinates
- With the sub-keyword **var_extents** portions of variables are made addressable during post-processing. In the same way multi-dimensional variables are supplied with indices in the simulation model the model they also have an index description in the model output description file for purposes of model output post-processing. It is advisable, that these two descriptions coincide. The index range is described by a start and an end index vi_ext_{i1} and/or vi_ext_{i2}.
  Index set is a strictly increasing, equidistant set of integer values, index increment is 1,
  $vi\_ext_{i1} < vi\_ext_{i2}$,
  $vi\_ext_{i1} \leq 0$ is possible.
- Coordinate values and index values are assigned in a one-to-one manner.
- For multi-dimensional variables that do not exist on an assigned grid completely or partially, simply assign formal coordinate axes to.
- Specify at least one model output variable in <model>.mdf.

SimEnv data type			Description	Restriction
byte	or	int*1	1 byte integer	not for Python models
short	or	int*2	2 bytes integer	not for Python models
int	or	int*4	4 bytes integer	
float	or	real*4	4 bytes real	
double	or	real*8	8 bytes real	not for Python models

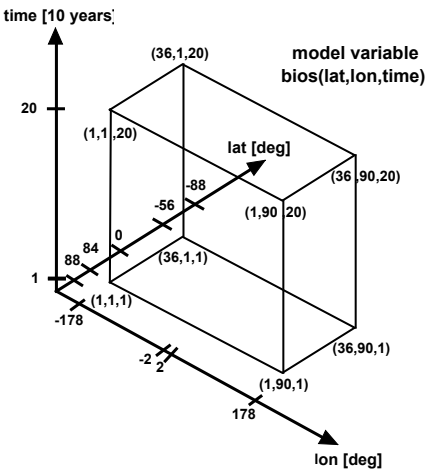
Tab. 5.2 *SimEnv data types*

For the following example of a model output description file and the assigned grid for model variable bios check Example 1.1 at page 4:

```
mdf                 descr         World with a resolution of
mdf                 descr         4° lat    x 4° lon x
mdf                 descr         4 levels  x 20 time steps
mdf                 descr         Data centred per lat-lon cell
mdf                 descr         This file is valid for all models
mdf                               world_[ f | c | cpp | py | sh ]

coordinate   lat    descr         geographic latitude
coordinate   lat    unit          deg
coordinate   lat    values        equidist_end 88(-4)-88

coordinate   lon    descr         geographic longitude
coordinate   lon    unit          deg
coordinate   lon    values        equidist_end -178(4)178

coordinate   level  descr         atmospheric vertical level
coordinate   level  unit          level no
coordinate   level  values        list 1,7,11,16

coordinate   time   descr         time in decades
coordinate   time   unit          10 years
coordinate   time   values        equidist_nmb 1(1)20

variable     atmo   descr         aggregated atmospheric state
variable     atmo   unit          without
variable     atmo   type          float
variable     atmo   coords        lat  , lon  , level , time
variable     atmo   var_extents   1:45 , 1:90 , 1:4   , 1:20

variable     bios   descr         aggregated biospheric state
variable     bios   unit          g/m2
variable     bios   type          float
variable     bios   coords        lat    , lon      , time
variable     bios   coord_extents 84:-56 , -178:178 , 1:20
variable     bios   var_extents   1:36   , 1:90     , 1:20

variable     atmo_g type          int
variable     atmo_g coords        time
variable     atmo_g var_extents   1:20

variable     bios g type          int
```

Example-file: world_[f | c | cpp | py | sh].mdf

Example 5.1 *Model output description file <model>.mdf*



Definition of model variable bios refers to Example 5.1 above.
The triples at the edges of the grid are the indices of model variable bios(lat,lon,time) for the appropriate grid cells.

Fig. 5.2 *Model variable definition: Grid assignment*

5.3 Model Interface for Fortran and C/C++ Models

Tab. 5.3 describes the functions that can be used in user models written in Fortran or C/C++ to adjust experiment targets for the current single run of the run ensemble and to output model results from the current single run. Two additional functions are responsible to initialize and/or finish SimEnv model coupling interactions.
Finally, two other model coupling functions are available: One function can be used to get the number of the current single run and an other to announce output of a slice of the data of a defined model variable. The latter is good for models with multi-dimensional variables where at least one dimension is omitted in the model's variable declaration because the dynamics for this dimension is calculated in place (e.g., time). The assigned variable then has a lower dimensionality than the corresponding variable in the model output description file. Nevertheless the slice-function ensures that model output over the omitted dimension can be handled in model output post-processing in common.

Model coupling functions are generic. To distinguish between the programming languages function names have a language suffix _f (for Fortran) and _c (for C/C++). All functions have a 4-byte integer function value (integer*4 and/or int). Implementation of the functions for C/C++ is based on a call by reference for the function arguments.

In Tab. 5.3 input and output data types are documented for functions used in Fortran. For C/C++ the corresponding data types are valid.

Function name without suffix _f, _c	Function description	Arguments / function value	Argument / function value description
simenv_end ( )	close model coupling interface **Apply always after the last call of the other SimEnv functions in the model**	integer*4 simenv_end (function value)	return code = 0 ok = 2 I/O error for model output file
simenv_get (target_name, target_val_def, target_val_adj)	gets the numerical modification (adjustment) for the target (parameter / initial value / boundary value) to be experimented with in the current single run	character*(*) target_name (input)	name of the target in <model>.edf
		real*4 target_val_def (input)	nominal / default (non-adjusted) target value. If target_name is not defined in <model>.edf then target_val_adj is set to target_val_def
		real*4 target_val_adj (output)	adjusted target value
		integer*4 simenv_get (function value)	return code = 0 ok = 1 target_name undefined: target_val_adj := target_val_def = 3 warning w.r.t. target_val_def or adjustment (check Tab. 6.6 at page 39)
simenv_get_run (run_int, run_char)	gets run number of current run as an integer value and a character string	character*6 run_char (output)	current run number with leading zeros
		integer*4 run_int (output)	current run number
		integer*4 simenv_get_run (function value)	return code = 0 ok
simenv_ini ( )	initialize model coupling interface **Apply always before the first call of the other SimEnv functions in the model**	integer*4 simenv_ini (function value)	return code = 0 ok = 2 I/O error for model output file = 3 error memory allocation = 4 I/O error for <model>.edf_bin = 5 I/O error for <model>.mdf_bin = 6 I/O error for <model>.edf_adj = 7 wrong single run number
simenv_put (var_name, field)	outputs model results to native SimEnv output file(s)	character*(*) var_name (input)	name of the variable in <model>.mdf to be output
		dimension field(...), type according to <model>.mdf (input)	data of variable var_name to be stored as simulation results
		integer*4 simenv_put (function value)	return code = 0 ok = 1 var_name undefined = 2 I/O error model output file

Function name without suffix _f, _c	Function description	Arguments / function value	Argument / function value description
simenv_slice (var_name, idim, ifrom, ito)	announces to output at next simenv_put call only a slice of variable var_name. This announcement becomes inactive after performance of the appropriate simenv_put	character*(*) var_name (input)	name of the variable in <model>.mdf to be sliced
		integer*4 idim (input)	dimension to be sliced
		integer*4 ifrom (input)	slice to start at position ifrom
		integer*4 ito (input)	slice to end at position ito
		integer*4 simenv_slice (function value)	return code = 0 ok = 1 var_name undefined = 3 inconsistency between variable and idim, ifrom, ito = 4 slice storage exceeded = 5 warning: slice overwritten

Tab. 5.3 *Model coupler functions at language level*

- Make sure consistency of type and dimension declarations between the model variables in model source code and the corresponding variable declarations in the model output description file <model>.mdf.
- Model variables that are not output completely or partially within the user model are handled in result-post-processing as their corresponding nodata-values (see Tab. 10.12).
- Application of simenv_slice_* for NetCDF model output may result in a higher consumption of computing time for each single run of the experiment compared with NetCDF model output without simenv_slice_*. For this case, keep in mind the trade-off between the demand for computing time and the demand for main memory.

In Example 15.1 at page 111 the model world_f.f and in Example 15.2 at page 113 the model world_c.c are explained.

5.4 Model Interface for Python Models

Due to the special features of Python the coupling interface to SimEnv differs from that for Fortran and C/C++ in chapter 5.3. Additionally, Python supports only some data types (check Tab. 5.2). Tab. 5.4 summarizes the coupling functions to use for a Python model.

Function name	Function description	Arguments / function value	Argument / function value description
simenv_ end_py ( )	close model coupling interface		
simenv_ get_py (target_name, target_def_val))	gets the numerical modification (adjustment) for the target (parameter / initial value / boundary value) to be experimented with in the current single run	string target_name (input)	name of the target in <model>.edf
		float target_val_def	nominal / default (non-adjusted) target value. If target_name is not defined in <model>.edf then target_val_adj is set to target>_val_def
		float get_py (function value)	adjusted target value target_val_adj
simenv_ get_run_py ( )	gets run number of current run as a character string	string get_run_py (function value)	current run number as string of the length 6 with leading zeros. If an error occurred then run_char = '------'
simenv_ ini_py ( )	initialize model coupling interface **Apply always before the first call of the other SimEnv functions in the model**	string ini_py (function value)	return code of the spawn function for a SimEnv executable
simenv_ put_py (var_name, field)	outputs model results to native SimEnv output file(s)	string var_name (input)	name of the variable in <model>.mdf to be output
		declaration of field(...) according to <model>.mdf (input)	data of variable var_name to be stored as simulation results. Maximum length of field is limited to 12.000 characters.
		put_py (function value)	unused
simenv_ slice_py (var_name, idim, ifrom, ito)	**Currently not available for Python models**		

Tab. 5.4 *Model coupler functions for Python models*

- Python coupling functions are declared in the file $SE_HOME/simenv.py. To use these functions in a Python model import it by
  from simenv import *
  and refer to it for example by simenv.get_run_py.
- For interfacing a Python model to SimEnv extend your operating system environment variable PYTHONPATH by $SE_HOME.
- Errors that occur during performance of one of the above functions are directly reported to <model>.mlog.

In Example 15.4 at page 116 the model world_py.py is described in detail.

5.4.1 Standard User-Defined Files for Python Models

<model>.ini
<model>.ini (see chapter 7.1 at page 43) is for Python models a mandatory script and has to have the same contents for all Python models:

```
$SE_HOME/py_model_ini
rc_py_model_ini = $?

# additional user-model specific commands can be implemented up from here
if test $rc_py_model_ini = 0
then
    ...
fi

exit $rc_py_model_ini
```

For an experiment restart (check chapter 7.2 at page 45) <model>.ini has to be performed again. To force this specify in <model>.cfg (check chapter 10.1at page 81) for the sub-keyword restart_ini the value yes.

5.5 Model Interface at Shell Script Level

For models that do not allow to implement the model coupling interface at programming language level (e.g., because source code is not available) SimEnv supplies a coupling interface at shell script level: the shell script <model>.run (see chapter 7.1 at page 43) is used to wrap the model and optionally to have at disposal corresponding functionality of the SimEnv coupling functions of Tab. 5.3).

- For the model interface at the shell script level, i.e., within the shell script <model>.run the adjusted experiment targets for the current single run from the whole run ensemble can be made available within <model>.run to forward them by any means the modeller is responsible for to the model under investigation.
  One common way to forward experiment targets to the model is to place current numerical target values as arguments to the model at model command line. Another way could be to read the targets from a special file in a special file format.
- Directly before performing simenv_get_sh make sure that the shell script variables target_name and target_def_val have be specified. At the end of each simenv_get_sh these variables are set again to empty strings.
- After running . $SE_HOME/simenv_get_sh an experiment target <target_name> from the experiment description file <model>.edf is available in <model>.run as a shell script variable <target_name> and the adjusted value of the target is available as $<target_name>.
- After running the model model output has to be identified and potentially transformed within <model>.run for SimEnv output. To do this simply write your own simenv_put_sh as a transformation program that reads in all the native model output and outputs it to SimEnv by applying the coupling functions simenv_*_* from the SimEnv model coupler at language level.

- Tab. 10.10 lists the built-in (pre-defined) shell script variables that are used in $SE_HOME/simenv_*_sh and finally in <model>.run.

Command name	Command description	Arguments	Argument description
. $SE_HOME/ simenv_end _sh	wrap up current single run **Apply always as the last command in <model>.run**		
target_name= '...' target_def_val= ... . $SE_HOME/ simenv_get _sh	gets a numerical modification (adjustment) for the target (parameter / initial value / boundary value) to be experimented with in the current single run	script variable target_name (input)	name of the target in <model>.edf
		script variable target_def_val (input)	nominal / default (non-adjusted) target value. If target_name is not defined in <model>.edf then target_val_adj is set to target>_val_def
		script variable target_name (output)	shell script variable with the same name as the value of target_name. Script variable value is the adjusted target value target_val_adj.
. $SE_HOME/ simenv_get_run _sh	gets run number of current run as an integer and a character script variable	run_char (output)	shell script variable with the current run number with leading zeros
		run_int (output)	shell script variable (type integer) with the current run number
. $SE_HOME/ simenv_ini _sh	initialize current single run **Apply always as the first command in <model>.run**	SE_RUN (output)	operating system environment variable SE_RUN is set to the current run number of the simulation experiment
. $SE_HOME/ simenv_put _sh	**Not available at shell script level**		**Write your own simenv_put_sh at the language level using the SimEnv coupling functions from Tab. 5.3 or Tab. 5.4**
. $SE_HOME/ simenv_slice _sh	**Not available at shell script level**		

Tab. 5.5 *Model coupler functions at shell script level*

In Example 15.5 at page 117 the model shell script world_sh.run is described in detail.

```
. $SE_HOME/simenv_ini_sh

# get adjusted value for the a target p_def, defined in the edf-file
target_name='p_def'
target_def_val=2.
. $SE_HOME/simenv_get_sh
# now shell script variable p def          is   available
# value of shell script variable p_def     is   according to edf-file

# get adjusted value for a target p_undef, not defined in edf-file
```

```
target_name='p_undef'
target_def_val=-999.
. $SE_HOME/simenv_get_sh
# now shell script variable p_undef        is   available
# value of shell script variable p_undef   is   -999.

# ...

. $SE_HOME/simenv_end_sh
```

Example file: world_sh.run

Example 5.2 *Addressing target names and values for the model interface at shell script level*

5.6 Model Interface for GAMS Models

SimEnv allows to couple GAMS models to the experiment shell. A GAMS model for SimEnv can consist of a GAMS main model and GAMS submodels.
Therefore, two additional include-statements have to be inserted into these GAMS model source code files where experiment targets are to be adjusted or model variables are to be output. GAMS model source code files to be interfaced to SimEnv are one GAMS main model and a number of GAMS sub-model that are called directly from the main model. All these GAMS model source code files have to be located in the current working directory. Additional GAMS sub-programs (included files) are not affected bei SimEnv, but you should keep in mind that the GAMS code within SimEnv will be executed in a subdirectory of the current working directory (see below) and so the include statements have to be changed, if the files are adressed in a relative manner (see below).

The include files are

- $include <GAMS_model>_simenv_get.inc
  $include <GAMS_model>_simenv_put.inc
  where <GAMS_model> is the name of a GAMS model file without extension .gms under consideration.
  The include statement $include <GAMS_model>_simenv_put.inc has to be placed in the GAMS model file at such a position where all the variables from the model output description file can be output by GAMS put-statements.
- During experiment preparation the file **<GAMS_model>_simenv_put.inc** and during experiment performance files **<GAMS_model>_simenv_get.inc** are generated automatically to forward GAMS model output to SimEnv data structures and to adjust investigated experiment targets, respectively.
  These include files correspond with the simenv_put and simenv_get functions at the language level (see chapter 5.3).
- For interfacing a GAMS model to SimEnv extend your operating system environment variable PYTHONPATH by $SE_HOME.
- In the course of experiment preparation the GAMS model and all sub-models that are specified in <model>.gdf (see below) are transformed automatically. Each GAMS model single run from the run ensemble is performed in a separate sub-directory of the current working directory. Transformed GAMS models and sub-models are copied to this sub-directory and are performed from there. Keep this in mind if you specify in any GAMS model include statements with relative paths.

In Example 15.6 at page 119 the model gams_model.gms is described in detail.

Additionally, the following settings are valid:

- An ASCII GAMS description file **<model>.gdf** (see below) has to be supplied to specify the GAMS submodels and assigned targets and model variables in detail.
- Maximum dimensionality of any model output variable declared in <model>.mdf is 4 for GAMS models.

Note the following information:

- To output the GAMS model status to SimEnv a
  PARAMETER modstat
  has to be declared and the statement
  modstat = <model_name>.modelstat
  has to be incorporated in the GAMS model before the $include <GAMS_model>_simenv_put.inc line. The variable modstat has to be stated in the model output description file <model>.mdf and the GAMS description file <model>.gdf.
- Relevant information is output to the model log file <model>.mlog.

5.6.1 Standard User-Defined Files for GAMS Models

<model>.ini
<model>.ini (see chapter 7.1 at page 43) is for GAMS models a mandatory script and has to have the contents for all GAMS models:

```
$SE_HOME/gams_model_ini
rc_gams_model_ini = $?

# additional user-model specific commands can be implemented up from here
if test $rc_gams_model_ini = 0
then
    ...
fi

exit $rc_gams_model_ini
```

For an experiment restart (check chapter 7.2 at page 45) <model>.ini has to be performed again. To force this specify in <model>.cfg (check chapter 10.1at page 81) for the sub-keyword restart_ini the value yes.

<model>.run
<model>.run (see chapter 7.1 at page 43) has for each GAMS model the same contents:

```
. $SE_HOME/simenv_ini_sh
$SE_HOME/gams_model_run
. $SE_HOME/simenv_end_sh
```

<model>.end
<model>.end (see chapter 7.1 at page 43) is for GAMS models a mandatory script and has to have the contents for all GAMS models:

```
$SE_HOME/gams_model_end

# additional user-model specific commands can follow
```

Python script language is used to prepare, run and to end a GAMS model.

<model>.edf
Corresponding experiment targets in the experiment description file <model>.edf (see chapter 0 at page 33) and in the GAMS model source code must have same names. In the GAMS model code the targets specified in the experiment description file have to be of type PARAMETER and have be defined before the include statement $include simenv_get.inc.

<model>.mdf
Corresponding variables in the model output description file and in the GAMS model source code must have same names. The variable type has to be always real*4 / float in the model output description file. In GAMS model code the model variables declared in the model output description file can be of the numeric types VARIABLES or PARAMETER. Currently, dimensionality of GAMS model output is limited to 0, 1 or 2.

With respect to Example 15.6 the model output description file could look like

```
coordinate   plant     descr          canning plants
coordinate   plant     unit           plant number
coordinate   plant     values         equidist_end 1(1)2

coordinate   market    descr          canning markets
coordinate   market    unit           market number
coordinate   market    values         equidist_end 1(1)3

variable     a         descr          plant capacity
variable     a         unit           cases
variable     a         type           float
variable     a         coords         plant
variable     a         var extents    1:2

variable     x         descr          shipment quantities
variable     x         unit           cases
variable     x         type           float
variable     x         coords         plant , market
variable     x         var_extents    1:2   , 1:3

variable     z         descr          total transportation costs
variable     z         unit           10^3 US$
variable     z         type           float

variable     modstat   descr          model status
variable     modstat   type           float
```

Example file: gams_model.mdf

Example 5.3 *Model output description file for a GAMS model*

5.6.2 GAMS Description File <model>.gdf

The ASCII GAMS description file <model>.gdf is intented to create a block of lines for each GAMS submodel with a simenv_get.inc file and/or a simenv_put.inc file. The block holds the specific characteristics of GAMS model input and output needed by SimEnv to generate GAMS put-statements. All model variables from the model output description file and all targets from the target description file have to be used in this file again.

<model>.gdf is an ASCII file that follows the coding rules in chapter 11 at page 91 with the keywords, names, sub-keywords, and info as in Tab. 5.1.

<keyword>	<name>	<sub-keyword>	Line type	Max. num-ber of lines	<info>	Explanation
gdf	<nil>	descr	o	any	<string>	GAMS coupling description
		keep_runs	o	1	<value_list>	value list of run numbers where single GAMS model runs are to be stored by keeping their corresponding sub-directories For syntax see Tab. 11.4
		options	o	1	<string>	string of options, GAMS is started with from command line
model	model_name (without extension .gms)	descr	o	1	<string>	(sub-)model description
		type	m	1	[main | sub]	identifies GAMS main or submodel
		get	m	exactly number of targets	<target_name>	get resulting adjustment for <target_name> to this model
		put	m	exactly number of model vari-ables	(<var_name> {.<suffix_set>} {(<index_set>)}) {<format>}	put values of SimEnv model output variable <var_name> from this model to SimEnv output. GAMS variable <var_name> has the specified suffix and index sets and is output ac-cording to <format>

Tab. 5.6 *Elements of a GAMS description file <model>.gdf*

To Tab. 5.6 the following additional rules and explanations apply:

- For the description of **line type** check Tab. 11.3 at page 92.
- Each target and each model variable as declared in <model>.edf and <model>.mdf respectively has to be used in the <info>-field of <model>.gdf exactly one time.
- To each GAMS model <model_name> an arbitrary number of targets and model variables can be assigned to by the corresponding get and/or put sub-keyword.
  To each sub-model (type = sub) at least one get or one put sub-keyword must be assigned to. The main model (type = main) can be configured without any get and put keywords. This is useful when the main model simply calls sub-models.
- Each model <model_name> in <model>.gdf with at least one get sub-keyword has to have an $include <model_name>_simenv_get.inc statement in the corresponding GAMS model file <model_name>.gms
- Each model <model_name> in <model>.gdf with at least one put sub-keyword has to have an $include <model_name>_simenv_put.inc statement in the corresponding GAMS model file <model_name>.gms
- There has to be exactly one main GAMS model, identified by the <subkeyword> type. All other models have to be of type = sub.
- The <info>-field for the <sub-keyword> put is adapted to GAMS syntax to output GAMS model variables. Afterwards this output is used to generate the appropriate SimEnv output.
  <index_set> is mandatory for variables with a dimensionality > 0. Otherwise, specification of <index_set> is forbidden. Indices as used in the GAMS model are separated from each other by comma.

With respect to Example 15.6 the GAMS description file could look like

```
gdf                 descr           GAMS model output description
gdf                 descr           for the examples in the SimEnv
gdf                 descr           User's Guide
gdf                 keep_runs       list 0,1

model   gams_model  descr           this is the only GAMS model to use
model   gams_model  type            main
model   gams_model  get             dem_ny
model   gams_model  get             dem_ch
model   gams_model  put             x.l(i,j):10:5
model   gams_model  put             a(i):10:5
model   gams_model  put             z.l
model   gams_model  put             modstat
```

Example file: gams_model.gdf

Example 5.4 *GAMS description file <model>.gdf*

If the model gams_model from the above Example 5.5 would be coupled with two additional sub-models sub_m1 and sub_m2 where both sub-models interact with SimEnv the GAMS description file could look like
(without taking into consideration plausibility with respect to model contents)

```
model   gams_model  type            main
model   gams_model  put             modstat

model   sub_m1      type            sub
model   sub_m1      get             dem_ny
model   sub_m1      put             x.l(i,j):10:5
model   sub_m1      put             a(i):10:5

model   sub_m2      type            sub
model   sub_m2      get             dem_ch
model   sub_m2      put             z.l
```

or

```
model   gams_model  type            main

model   sub_m1      type            sub
model   sub_m1      get             dem_ny
model   sub_m1      put             x.l(i,j):10:5
model   sub_m1      put             a(i):10:5

model   sub_m2      type            sub
model   sub_m2      get             dem_ch
model   sub_m2      put             z.l
model   sub_m2      put             modstat
```

Example 5.5 *GAMS description file for <model>.gdf*

5.6.3 Files Created during GAMS Model Performance

During experiment performance of a GAMS model each single run from the experiment is performed individually in a directory run<run_char> of the current working directory. Each directory is generated automatically before performing the corresponding single run and removed after perfomance of this single run. With the sub-keyword keep_runs the user can force to keep sub-directories for later check of the transformed model code.

Additionally to the files listed in Tab. 10.5, during the performance of a GAMS model the files <gams_model>_[pre | main | post].inc are created temporarily in the current working directory by <model>.ini and are deleted after the whole experiment where <gams_model> is a placeholder for the model of type main and all models of type sub in the gdf-file.

5.7 Distributed Models

SimEnv supports performance of distributed models. Distributed models may consist from a web or a chain of stand-alone sub-models, i.e., the model is computed by performing a set of stand-alone binaries. Each of these stand-alone sub-models can use SimEnv functionality, i.e., simenv_get_*, simenv_get_run_*, simenv_put_*, or simenv_slice_*. In such sub-models simenv_ini_* and simenv_end_* have to be incorporated in. Additionally, the corresponding SimEnv model coupling functionality at shell script level (simenv_*_sh modules) can be applied. The model description file <model>.mdf collects all the state variables from all sub-models and the experiment description file <model>.edf collects all the targets for all sub-models.

Sub-models can reside on different machines. The only prerequisite for this is that the current working directory and the model output directory can be mapped to each of this machines.

5.8 Running Coupled Models Outside SimEnv

To run a model coupled to SimEnv outside the simulation environment in its native mode as before code transformation the following simple rules have to be applied to the model:

- For Fortran and C/C++ models:
  Link the model with the object library $SE_HOME/libsimenvdummy.a instead of $SE_HOME/libsimenv.a. SimEnv function values (return codes) from this library are zero, function simenv_get_* forwards target_val_def to target_val_adj, simenv_get_run_* returns integer run 0 and character run string ‘ ‘ (six blanks).

- For Python models:
  Replace in the model source code
  ```
  from simenv import *
  ```
  by
  ```
  from simenvdummy import *
  ```
  function values (return codes) from simenvdummy.py are zero, dummy function simenv_get_py forwards target_val_def to target_val_adj, simenv_get_run_py returns run 000000.

- For GAMS models:
  Handle in the model source code
  ```
  $include <model>_simenv_get.inc
  ```
  and
  ```
  $include <model>_simenv_put.inc
  ```
  as comment lines.

6 Experiment Preparation

Experiment preparation is the first step in experiment performance of a model coupled to the environment. In an experiment description file <model>.edf all information to the selected experiment type and its numerical equipment is gathered in a structured way.

6.1 Experiment Description File <model>.edf

<model>.edf is an ASCII file that follows the coding rules in chapter 11 at page 91 with the keywords, names, sub-keywords, and info as in Tab. 6.1.

<keyword>	<name>	<sub-keyword>	Line type	Max. number of lines	<info>	Explanation
edf	<nil>	descr	o	any	<string>	experiment description
		type	m	1	[behaviour | sensitivity | monte-carlo]	experiment type
target	target_ name	descr	o	1	<string>	target description
		unit	o	1	<string>	target unit
		type	m	1	see Tab. 6.2	adjustment type
		default	m	1	<value>	target default value <target_def_val>
		adjusts	c3	1	<experiment-specific>	experiment-specific information
specific	<nil>	<experiment-specific>	m	<experiment-specific>	<experiment-specific>	experiment-specific information

Tab. 6.1 *Elements of an experiment description file <model>.edf*

To Tab. 6.1 the following additional rules and explanations apply:

- For the description of **line type** check Tab. 11.3 at page 92.
- Target names must differ from model variables and coordinate names in the model output description file (see chapter 5.1) and from built-in and user-defined operator names for model output post-processing (see chapter 8.6.2).
- A target name is the symbolic parameter / driver / initial value / boundary value name, corresponding to targets of the investigated model. Correspondence is achieved by applying the SimEnv model coupling function simenv_get_* in the model.
- **The default value as specified in <model>.edf and not the default value from the model code is used to derive the adjusted value.**
- All experiment-specific information is explained in the appropriate chapters.
- Specify at least one experiment target.
- When preparing an experiment an experiment input file <model>.edf_adj is generated with the values to be finally used for the resulting adjustments. These values are applied to the default values of the targets according to the specified adjustment type (see Tab. 6.2 below) before finally influencing the dynamics of the model. The sequence of elements (columns) of each record of <model>.edf_adj corre-

sponds with the sequence of targets in the target name space (see chapter 11 at page 91), the sequence of records corresponds with the sequence of single model runs of the experiment. For each experiment a single model run with run number 0 is generated automatically as the nominal run of the model without adjustments. This run does not have an assigned record in <model>.edf_adj.

Adjustment type	Meaning
set	value setting: Use the adjustment to the target default value within the SimEnv function simenv_get_* as the final adjusted value. Not available for local sensitivity analysis
add	addition: Add the declared adjustment to the target default value within the SimEnv function simenv_get_* to get the final adjusted value to use.
multiply	multiplication: Multiply the declared adjustment with the target default value within the SimEnv function simenv_get_* to get the final adjustment to use. Differing implementation for local sensitivity analysis (check chapter 6.4.1).

Tab. 6.2 *Adjustment types in experiment preparation*

6.2 Behavioural Analysis

The experiment-specific information for experiment description files in Tab. 6.1 at page 33 is defined for behavioural analysis as follows:

<keyword>	<name>	<sub-keyword>	Line type	Max. number of lines	<info>	Explanation
target	target_ name	adjusts	a	1	<value_list>	value list of target value adjustments <adj_val> to apply. For syntax see Tab. 11.4
specific	<nil>	comb	m	1 or any	[default | <combination> | file {<path>/} <file_name>]	information how to scan the spanned target space

Tab. 6.3 *Experiment-specific elements of an edf-file for behavioural analysis*

To Tab. 6.3 the following additional rules and explanations apply:

- For the description of **line type** check Tab. 11.3 at page 92.
- For <sub-keyword> = comb the following rule yields:
  <info> = [default | <combination>] for available <sub-keyword> = adjusts
  <info> = [file {<path>/}<file_name>] for unavailable <sub-keyword> = adjusts
- Values of a value list have to be unique for available <sub-keyword> = adjusts and each target
  Assigned values from file {<path>/}<file_name> can be multiple defined for each target.
- <path> must not contain environment variables from operating system level.

6.2.1 Adjustments

Adjustment type	Set	Add	Multiply
adjusted target value =	<adj_val>	<target_def_val> + <adj_val>	<target_def_val> * <adj_val>

6.2.2 The Combination

- The combination **<combination>** defines the way in which the space spanned by the experiment targets will be inspected by SimEnv: This is done by applying operators „***“** and „**,“** to all stated experiment targets.
  - **The operator „*“** combines adjustments of different targets and so their resulting values combinatorially (“for all mesh points in a grid”).
    Compare with experiment description file (a) from Example 6.1 below.
  - **The operator „,“** combines adjustments of different targets and so their resulting values parallel (“on the diagonal”).
    For the operator „,“ the targets must have the same number of adjustments.
    Compare with experiment description file (b) from Example 6.1 below.
  - The operator „,“ has a higher priority than the operator „*“. Parentheses are not allowed:
    For example, p1 * p2 , p3 * p4 always combines p2 and p3 in parallel and this combinatorially with p1 and p4. A parallel combination of p1 * p2 with p3 * p4 by (p1 * p2) , (p3 * p4) is not possible.
    Compare with experiment description file (c) from Example 6.1 below.
  - In <combination> each target has to be used exactly once.
- By the default combination **default** all experiment targets are combined combinatorially.
  - comb default of the experiment description file (a) from Example 6.1 below is equivalent to comb p1 * p2 .
- Specification of **file** is only allowed for unused adjusts-sub-keywords all over the edf-file.
  - The adjustments are read from the adjustment data file **{<path>/}<file_name>**.
  - All targets are assumed to be combined in parallel. Each record of the data file represents one simulation run. The sequence of the adjustments (sequence of columns) in each record corresponds with the sequence of the targets in the target name space (see chapter 11 at page 91).
  - Syntax rules for value lists at page 91 yield.
  - Identical adjustments for a target are allowed.
  - During model output post-processing restricted capabilities for the operator behav apply for this experiment layout.
  - Compare with experiment description file (d) from Example 6.1 below. Combination is implicitly as comb p1 , p2. Experiment description files (b) and (d) in Example 6.1 below describe the same experiment.

6.2.3 Example

The first three experiment description files (a) to (c) represent an experiment description according to Fig. 4.3 (a) to (c) at page 12.

Results in values ...

(a)
```
edf              descr      Experiment description for the examples
edf              descr      in the SimEnv User's Guide (Fig.  4.3 (a))
edf              type       behaviour
target    p1     descr      parameter p1
target    p1     unit       without
target    p1     type       add
target    p1     default    1.
target    p1     adjusts    list 1, 3, 7, 8              ... 2,4,8,9 for p1
```

```
    target     p2    descr       parameter p2
    target     p2    unit        without
    target     p2    type        multiply
    target     p2    default     2.
    target     p2    adjusts     list 1, 2, 3, 4              ... 2,4,6,8 for p2

    specific         comb        default

(b) edf              descr       Fig.  4.3 (b)
    edf              type        behaviour
    target     p1    type        multiply
    target     p1    default     1.
    target     p1    adjusts     list 1, 3, 7, 8              ... 1,3,7,8 for p1
    target     p2    type        multiply
    target     p2    default     2.
    target     p2    adjusts     equidist_end 1(0.5)2.5       ... 2,3,4,5 for p2
    specific         comb        p1,p2

(c) edf              descr       Fig.  4.3 (c)
    edf              type        behaviour
    target     p1    type        set
    target     p1    default     1.
    target     p1    adjusts     list 1, 3, 7, 8              ... 1,3,7,8 for p1
    target     p2    type        set
    target     p2    default     2.
    target     p2    adjusts     equidist_end 1(1)4           ... 1,2,3,4 for p2
    target     p3    type        multiply
    target     p3    default     3.
    target     p3    adjusts     list 1.1, 1.5, 2.4           ... 3.3,4.5,7.2 for p3
    specific         comb        p2,p1*p3

(d) edf              type        behaviour          file world.dat d:
    target     p1    type        multiply             1     0
    target     p1    default     1.                   3     1
    target     p2    type        add                  7     2
    target     p2    default     2.                   8     3
    specific         comb        file world.dat_d           ... (1,2),(3,3),(7,4),(8,5)
                                                            ... for (p1,p2)

                                                 Example files: world.edf_a to world.edf_d
```

Example 6.1 *Experiment description file <model>.edf for behavioural analysis*

6.2.4 Experiment Performance

- Firstly, a model run 000000 with the default values of the experiment targets is performed.
- According to the keyword comb the appropriate runs are generated.
- The sequence of the runs corresponds with the sequence of the adjustments in the ASCII file <model>.edf_adj (check chapter 0 at page 33 for more information).

6.3 Monte-Carlo Analysis

The experiment-specific information for experiment description files in Tab. 6.1 at page 33 is defined for Monte-Carlo analysis as follows:

<keyword>	<name>	<sub-keyword>	Line type	Max. number of lines	<info>	Explanation
target	target_ name	adjusts	m	1	[<distribution> | file {<path>/} <file_name>]	distribution and distribution parameters to be applied for the target or import of an external sample <distr_val> from <file_name>
		sample	c4	1	[random | latin hypercube]	sampling strategy: random or latin hypercube sampling LHS
specific	<nil>	runs	m	1	<nr_of_runs>	number of runs > 10 to be performed for the experiment

Tab. 6.4 *Experiment-specific elements of an edf-file for Monte-Carlo analysis*

To Tab. 6.4 the following additional rules and explanations apply:

- For the description of **line type** check Tab. 11.3 at page 92.
- <distribution> = <distr_shortcut> (<distr_param_1> { , <distr_param_2> }) (check Tab. 6.5)
- For implicitly specified distributions according to Tab. 6.5 adjustments are applied to the specified distribution parameters of the distributions. Afterwards, a sample <distr_val> is generated from the distribution with the adjusted distribution parameters. Adjustment types add and multiply are not applied to the distribution parameter <distr_param> = standard deviation. Instead, the specified standard deviation from the experiment description file is used (adjustment type set is applied).
- For explicitly specified samples of any distribution by the ASCII file <file_name> adjustments are applied directly to the sample values <distr_val> from the file. For syntax rules for files check chapter 11. Each record of the ASCII file can hold only one sample value. Sample size has to be identical to <nr_of_runs> from the specific-keyword.
- In random sampling, there is no assurance that sampling points will cover all regions of the selected distribution. With Latin hypercube sampling LHS this shortcoming is reduced: The sampling range of the target is divided into <nr_of_runs> intervals of equal probability according to the selected distribution and from each interval exactly one sampling point is drawn. For more information on LHS check Fig. 6.1 below and see Imam & Helton (1998) and Helton & Davis (2000).
- The number of runs <nr_of_runs> must be greater than 10.

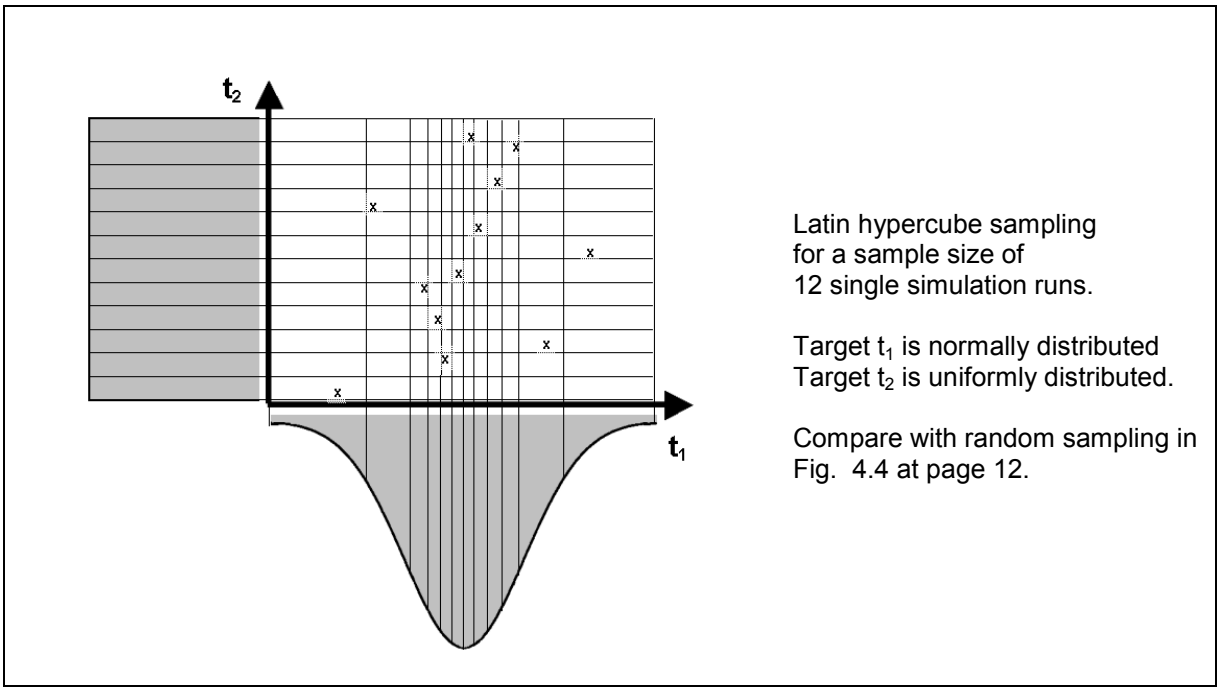


Fig. 6.1 *Monte Carlo analysis: Latin hypercube sampling*

6.3.1 Adjustments

Adjustment type	Set	Add	Multiply
for distribution: adjusted distr_param =	<distr_param>	<target_def_val> + <distr_param>	<target_def_val> * <distr_param>
		not for standard deviation instead, adjustment type "set" is applied	
for file: adjusted target_value =	<distr_val>	<target_def_val> + <distr_val>	<target_def_val> * <distr_val>

6.3.2 Distribution Functions and their Parameters

Distribution function	<distr_ shortcut>	<distr_param_1>	<distr_param_2>	Restriction
uniform	U	lower boundary	upper boundary	lower boundary < upper boundary
normal	N	mean value	variance	variance > 0
lognormal	L	mean value of a normally distributed target	variance of a normally distributed target	variance > 0
exponential	E	mean value	---	mean value > 0

Tab. 6.5 *Probability density functions and their parameters*

For more information on the distribution functions see chapter 4.3 and Tab. 4.2.

6.3.3 Example

```
(e) edf           descr     Experiment description for the examples
    edf           descr     in the SimEnv User's Guide
    edf           type      Monte-Carlo


    target   p2   descr     parameter p1
    target   p2   unit      without
    target   p2   type      multiply
    target   p2   default   2.
    target   p2   sample    latin hypercube
    target   p2   adjusts   distr U(0.5,1.5)    p2 is a realization of a uniform distrib.
                                                 between 0.5*2 and 1.5*2

    target   p1   type      add
    target   p1   default   1.
    target   p1   sample    random
    target   p1   adjusts   distr N(0,0.4)      p1 is a realization of a normal distribution
                                                 with mean = 1+0 and variance = 0.4

    target   p3   type      add
    target   p3   default   3.
    target   p3   adjusts   file world.dat_e    realization of p3 is read from file
                                                 world.dat_e and afterwards 3 is added

    specific      runs      250
```

Example file: world.edf_e

Example 6.2 *Experiment description file <model>.edf for Monte-Carlo analysis*

6.3.4 Experiment Performance

- Firstly, a model run 000000 with the default values of the experiment targets is performed which represents the deterministic case.
- The sequence of the runs corresponds with the sequence of the adjustments in the ASCII file <model>.edf_adj. <model>.edf_adj is generated from random numbers of the appropriate distributions U(0,1), N(0,1), L(0,1), and/or E(1). For more information on <model>.edf_adj check chapter 0 at page 33.
- If the resulting distribution parameters do not fulfil the restrictions in Tab. 6.5 the following adaptations are applied

Distribution	Condition	Adaptation
U	lower boundary > upper boundary	boundaries are interchanged
U	lower boundary = upper boundary	lower boundary := lower boundary – 0.5 upper boundary := upper boundary + 0.5
E	mean < 0	mean := -mean
E	mean = 0	mean := abs(model default value) for model default value ≠ 0 1 else

Tab. 6.6 *Probability density functions: Distribution parameters - conditions and adaptation*

6.4 Local Sensitivity Analysis

The experiment-specific information for experiment description files in Tab. 6.1 at page 33 is defined for local sensitivity analysis as follows:

<keyword>	<name>	<sub-keyword>	Line type	Max. number of lines	<info>	Explanation
target	target_ name	adjusts	f	0		
specific	<nil>	incrs	m	1	<value_list>	increments <incr_val> for all targets defined by a value list For syntax see Tab. 11.4

Tab. 6.7 *Experiment-specific elements of an edf-file for local sensitivity analysis*

To Tab. 6.4 the following additional rules and explanations apply:

- For the description of **line type** check Tab. 11.3 at page 92.
- Values from the value list must be unique.
- **Note** that computation of adjusted values in local sensitivity analysis differs from all other experiment types.

6.4.1 Adjustments

Adjustment type	Set	Add	Multiply
adjusted target value =	undefined for this experiment type	<target_def_val> ± <incr_val>	<target_def_val> * (1 ± <incr_val>)

As an example, the linear sensitivity function (see chapter 4.4 at page 14) is then as follows:

for adjustment A lin = $\frac{z(def \pm incr) - z(def)}{incr}$

for adjustment M lin = $\frac{z(def * (1 \pm incr)) - z(def)}{def * incr}$

6.4.2 Example

```
(f) edf             descr      Experiment description for the examples
    edf             descr      in the SimEnv User's Guide
    edf             type       sensitivity

    target    p1    descr      parameter p1
    target    p1    unit       without
    target    p1    type       add
    target    p1    default    1.

    target    p2    type       multiply
    target    p2    default    2.

    specific        incrs      equidist_end 0.01(0.01)0.05
```

Example file: world.edf_f

Example 6.3 *Experiment description file <model>.edf for local sensitivity analysis*

6.4.3 Experiment Performance

- Each experiment target will be adjusted by the same increments as those stated in the incrs info-field
- Adjustment M with the default target value = 0 is indicated during performance of the simulation experiment by a warning message to the file <model>.mlog.
- For finite sensitivity functions several runs have to be performed:
  - A nominal run with the default values of the experiment targets (run number 000000)
  - Per target and per increment two runs with the default values of all targets except that one under consideration, where the adjustment is applied according to the above adjustment rules
  - Accordingly, the number of resulting runs is 2 * number_of_targets * number_of_increments + 1
- Results of each model run are stored and sensitivity functions are applied during model output post-processing.
  The following sensitivity functions can be performed:
  Linear, squared, absolute, relative as well as a symmetry test.
- The sequence of the simulation runs are determined in the following manner:

```
nominal run
loop        over increment sequence
            loop over experiment targets
                 adjustment for increment
                 adjustment for negative increment
            end loop
end loop
```

7 Experiment Performance

After experiment preparation experiment performance is the second step in running a model coupled to SimEnv. Each multi-run experiment can be performed sequentially or in parallel. Besides a new-start of an experiment a restart after an experiment interrupt or only for an experiment slice can be handled by SimEnv.

7.1 Experiment Start

- Currently an experiment can be performed sequentially on the login-machine and in parallel and/or sequential mode in a job class controlled by the LoadLeveler. In parallel mode the single runs of the run ensemble are distributed to all allocated nodes with their assigned processors. One communication processor is responsible for experiment management.
- The user can define an experiment preparation shell script **<model>.ini** that is performed additionally after standard experiment preparation when starting a new experiment. For experiment restart <model>.ini is performed only on request (see chapter 7.2 below).
  In <model>.ini additional settings / checks can be performed. For return codes unless zero from <model>.ini the experiment will not be started.). Make sure that <model>.ini has execute permission by chmod u+x. For Python and GAMS models <model>.ini is a mandatory script with pre-defined contents. Check chapters 5.4.1 and 5.6.1 for more information.
- The model to be applied within the SimEnv experiment has to be wrapped in the shell script **<model>.run**. <model>.run is performed for each single run within the run ensemble.
  - **Make sure that**
    - **. $SE_HOME/simenv_ini_sh is the first command and**
    - **. $SE_HOME/simenv_end_sh is the last command**

    in <model>.run (see Tab. 5.5 at page 26 and Example 7.2 below).
  - Ensure by chmod u+x that <model>.run has execute permission.
  - To cancel the whole experiment after the performance of the current single because of a any condition of the current single run make sure a file **<model>$run_char.err** exists as an indicator to stop. You can create this file in the model or in <model>.run. For the latter
    - Perform . $SE_HOME/simenv_get_run_sh to get the current run number <run_int> and <run_char> (see Tab. 5.5 at page 26 and Example 7.2 below).
    - Touch the file <model>$run_char.err.
  - For GAMS models <model>.run has a pre-defined structure. Check chapter 5.6.1 for more information.
- The model variables to be output during experiment performance are declared in the model output description file <model>.mdf
- The type and the targets of the experiment to be performed are declared in the experiment description file <model>.edf
- Mapping between experiment targets and targets in the model source code is achieved by application of the generic SimEnv function simenv_get_* in the model code or at shell script level.
- Output of model variables declared in <model>.mdf into SimEnv structures is achieved by the application of the generic SimEnv function simenv_put (and simenv_slice) in the model source code or by an appropriate, user-written module simenv_put_sh at shell script level.
- Model output from run number <run> is stored in the file <model>.out<run_char>.[nc | ieee] if the sum over all model output variables of a single run is less than the appropriate value specified in <model>cfg. Otherwise, model output from the complete experiment is stored in <model>.outall.[nc | ieee].
- For each experiment type a run number 0 with the default values of all experiment targets will be performed additionally to the runs declared in the experiment description file <model>.edf.
- During experiment performance a model log-file <model>.mlog is written where adjustments of experiment target values and possibly workarounds for wrong re-adjustments (only for experiment type Monte-Carlo analysis, see Tab. 6.6) are stored. All model output to the terminal is re-directed within SimEnv to the model protocol file <model>.mlog.

- During experiment performance an experiment log-file <model>.elog is written with the minutes of the experiment.
- Do not start and/or submit another experiment from a working directory where an experiment is still running.
- After the experiment has been finished the model-specific output from the experiment can be wrapped up with the optional shell script **<model>.end**.
- After the experiment has been finished an e-mail is send on demand (check chapter 10.1) to the address as specified in <model>.cfg.
- For more information check Fig. 7.1.

For the shell script world_*.ini the following contents could be defined:

```
# coarse 0.5° x 0.5° land-sea mask from file land_sea_mask.05x05
# in the current directory
# to a 4° x 4° resoluted land-sea-mask in file land_sea_mask.coarsed
# in the current directory to use for all single runs
land_sea_mask 4 4
rc_land_sea_mask=$?

# exit from world_*.ini with return code != 0
# as an indicator not to start the experiment
exit $rc_land_sea_mask
```

Example files: world_[f | c | cpp | py | sh].ini

Example 7.1 *Shell script <model>.ini for user-model specific experiment preparation*

For the shell script world_f.run the following contents could be defined:

```
# always perform at begin:
. $SE_HOME/simenv_ini_sh

# run the model:
world_f

# assuming a model return code != 0 as an indicator to stop
# the whole experiment for any reason.
# Touch the file below in the current working directory $SE_WD
# as an indicator to SimEnv for this.
if test $? -ne 0
then
      . $SE_HOME/simenv_get_run_sh
      touch $SE_WD/world_f.$run_char.err
fi

# always per at end:
. $SE_HOME/simenv_end_sh
```

Example file: world_f.run

Example 7.2 *Shell script <model>.run to wrap the user model*

For the shell script world_f.end the following contents could be defined:

```
# remove the file of the coarsed land-sea mask
rm –f land sea mask.coarsed
```

Example file: world_[f | c | cpp | py | sh].end

Example 7.3 *Shell script <model>.end for user-model specific experiment wrap-up*

7.2 Experiment Restart

When an experiment was interrupted / has failed because of any reason or in the case of partial experiment performance (see chapter 7.3 below) it can be restarted several times:

- Simply restart the experiment by simenv.rst without changing any of the SimEnv files describing the experiment and/or the model. The only exception may be the information for the experiment-keyword in the general model configuration file <model>.cfg.
- simenv.rst has the same usage as simenv.run
- Restart can be launched on an other machine / in an other job class than that of the interrupted experiment.
- Dependent on the experiment log-file <model>.elog, written in the interrupted experiment a single model run from the complete run ensemble in the restart experiment will be
  - Performed if this run has neither a start nor a finish information in the elog-file
  - Not performed if this run has a start as well as a finish information in the elog file
  - Performed anew if the run has a start information but no finish information in the elog-file.
    For this case a model restart shell script **<model>.rst** can be provided by the user optionally to prepare restart of this single model run (e.g., by deleting non-SimEnv temporary or output files). Make sure that <model>.rst has execute permission by chmod u+x.
    Make sure that . $SE_HOME/simenv_ini_sh is the first command in <model>.rst.
    After running $SE_HOME/simenv_get_run_sh the shell script variables run_int and run_char are available in <model>.rst (see above).
- Experiment restart works without standard SimEnv experiment preparation. Instead, experiment preparation files and other information from the interrupted experiment will be used.
- The optional experiment preparation shell script **<model>.ini** will be performed only on demand. This request is specified in the configuration file <model>.cfg with the sub-keyword restart_ini.
- **<model>.cfg** will be checked anew for experiment restart. Avoid to change information in <model>.cfg for a restart. The only exception is the information related to the experiment-keyword.
- Minutes of the restarted experiment will be appended to the <model>.mlog and <model>. elog files, respectively from the interrupted experiment.
- Restart can be applied to an experiment several times successively.
- Experiment restart can be performed also as partial experiments, independently on the partial status of the original model

For the model world_sh (check Example 15.5 at page 117) the following contents could be defined for the restart script world_sh.rst:

```
# always perform at begin
. $SE_HOME/simenv_ini_sh

# get run number
. $SE_HOME/simenv_get_run_sh
```

```
# remove all files from the temporary directory and the directory itself
if test -d run$run_char
then
   cd run$run_char
   rm -f *
   cd ..
   rmdir run$run_char
fi
```

Example file: world_sh.rst

Example 7.4 *Shell script <model>.rst to prepare model performance during experiment restart*

7.3 Experiment Partial Performance

- SimEnv enables to perform an experiment partially by performing only a run slice out of the whole run ensemble.
- Therefor assign appropriate run numbers to the corresponding experiment keywords in <model>.cfg.
- Make sure that begin run number and end run number represent run number from the experiment (including run number 0) and that begin run number ≤ end run number.
- A partial experiment performance is also possible for an experiment restart.
- For more information check Fig. 7.1.

7.4 Job Control for Experiment Performance at a Parallel Machine

- For experiment performance controlled by the parallel operating environment POE and the LoadLeveler make sure that the environment variable SE_HOME is set in your .profile-file correctly.
- On a login node to a parallel machine there is an additional SimEnv dialogue whether the experiment is to be submitted by POE and the LoadLeveler to a parallel or sequential job class of this parallel machine or is to be performed locally at the login node.
- Default job control files are supplied by SimEnv to ensure communication with POE and LoadLeveler. These job control files may be copied to the current working directory, can be modified and will then be used instead of the default job control files to start an experiment at a parallel or sequential job class.
  If necessary, copy **$SE_HOME/simenv.jcf_par** and/or **$SE_HOME/simenv.jcf_seq** to the current working directory SimEnv is started from, modify the file(s) according to the needs of the experiment you want to perform and / or the machine you want to use and start afterwards simenv.run (or simenv.rst). If available in the current working directory, these modified job control files are used instead of the original files in $SE_HOME.
  simenv.jcf_seq submits a job to a sequential batch class, simenv.jcf _par to a parallel batch class.
- Default job control files enable automatic restart of the experiment by the LoadLeveler after an interrupt of the job in a parallel or sequential job class caused by POE, the LoadLeveler or the operating system. The user does not need to restart the experiment manually after such an event.

7.5 Experiment-Related User Scripts and Files

Script / file	Explanation	Used for (*)	Exist status
Scripts (**)			
<model>.run	model shell script to wrap the model executable . $SE_HOME/simenv_ini_sh has to be the first command in <model>.run . $SE_HOME/simenv_end_sh has to be the last command in <model>.run Model coupler functions at shell script level can be applied in <model>.run Pre-defined contents for GAMS models (check chapter 5.5)	S R	mandatory
<model>.rst	model shell script to prepare single model run restart for such single runs that were started by not finished during the previous experiment start / restart . $SE_HOME/simenv_ini_sh has to be the first command in <model>.rst . $SE_HOME/simenv_get_run_sh can be applied in <model>.rst (check chapter 5.5)	R	optional
<model>.ini	model shell script to prepare simulation experiment additionally to standard SimEnv preparation Experiment will be not performed if return code from this script is unequal zero. For experiment re-start <model>.ini will be performed only on request.	S (R)	optional, for Python and GAMS models mandatory
<model>.end	model shell script to clean up simulation experiment from non-SimEnv files	S R	optional
Files			
<model>. <run_char>. err	touch this file in the model, in <model>.run and/or <model>.rst as an indicator to stop the complete experiment after single run <run_char> has been finished	A	optional
simenv.jcf_par	user-specific job control file to submit an experiment to a parallel class by the LoadLeveler Copy from $SE_HOME on demand	L	optional
simenv.jcf_seq	user-specific job control file to submit an experiment to a sequential class by the LoadLeveler. Copy from $SE_HOME on demand	L	optional

Tab. 7.1 *Experiment-related user scripts and files*

(*): script applied for

*S: **S**tart of an experiment by $SE_HOME/simenv.run <model>*

*R: **R**estart of an experiment by $SE_HOME/simenv.rst <model>*

file applied for

*L: **L**oadLeveler experiment submission*

*A: **A**ll experiment perform. at the login machine or by LoadLeveler submission*

(**): make sure the shell script has execute permission by chmod u+x

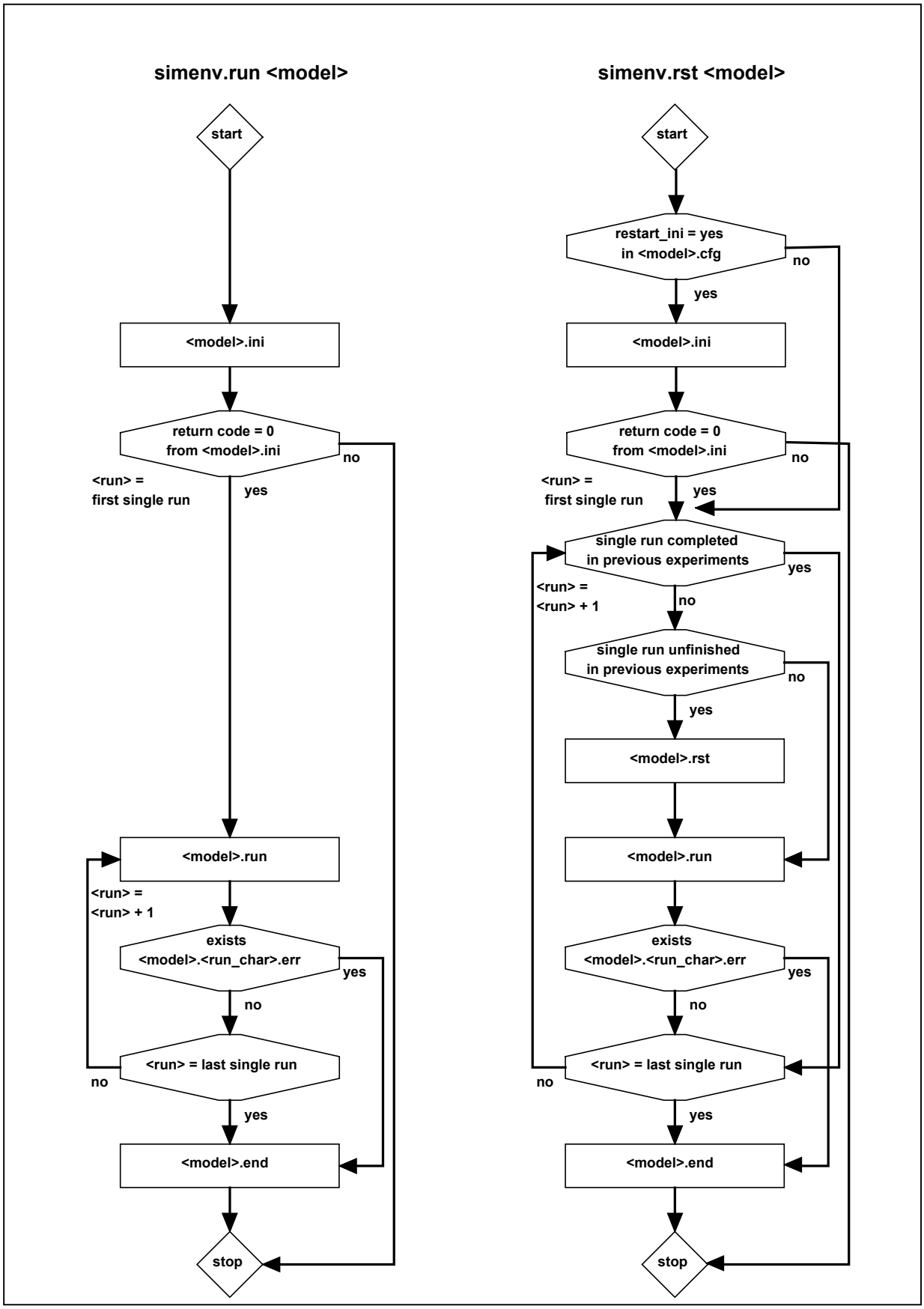


Fig. 7.1 *Flowcharts for performing simenv.run and simenv.rst*
First and last single run always refer to the corresponding settings in <model>.cfg

7.6 Saving Experiments

To save experiments for later use, e.g., by SimEnv post-processing, make sure to store from the experiment the following files:

- <mdel>.out[all | <run_char>].[nc | ieee] from the model output directory
- <model>.cfg from the current working directory
- <model>.mdf from the current working directory
- <model>.edf from the current working directory
- <model>.elog (optional) from the current working directory
- <model>:mlog (optional) from the current working directory

8 Experiment Post-Processing

Goal of post-processing is to navigate within the model / experiment output space by deriving interactively output functions / data that are to be visualized in experiment evaluation afterwards. Therefor SimEnv supplies operators that can be applied to model output and reference data. There are built-in basic and advanced operators and built-in experiment-specific operators. The user can define its own private operators and easily couple them to the post-processor. Operator chains and recursions are possible. Macros can be defined as abbreviations for operator chains.

8.1 Operands

Operands in expressions can be

- Model output variables (see below)
- Experiment targets
- Constants in integer or real*4 / float notation
- Character strings
- Operators
- Macros (see chapter 8.8)

To each operand (with the exception of character string operands) a

- Dimensionality **dim(operand)** and
  extents **ext(operand,i)** with i=1,...,dim(operand) and
  coordinates **coord(operand,i)** with i=1,...,dim(operand)
  are assigned to. The dimensionality is the number of dimensions, an extent is related to each dimension and represents the number of elements in that dimension. Extents are always greater than 1. To each dimension a coordinate is assigned to. Coordinates have a name and from all coordinate values the coordinate is defined for a subset is assigned to the extent of the dimension of the operand. Coordinate specification for operands follows that for model output variables. For more information see chapter 5.1.
- Operators transform dimensionality, dimensions, and coordinates of the their non-character operator arguments into unique dimensionality, dimensions and coordinates of the operator result (see chapter 8.3).
- Operands of dimensionality 0 do not have a coordinate assignment.
- Experiment targets and constants always have a dimensionality of 0.
- Consequently, a post-processor result as a sequence of operators applied to operands also has unique dimensionality, extents and coordinates.

8.2 Model Output Variables

- A variable of dimensionality n corresponds with a n-dimensional array and is defined at a n-dimensional grid, spanned up from the coordinate values of the assigned coordinates The complete data field of a model output variable or parts of it can be addressed in model output post-processing (see below). Dimensionality, dimensions and coordinate description of this data field is derived from the model variable description
- Model output variables are specified in the ASCII model output description file <model>.mdf by their
  - Name
  - Dimensionality
  - Extents
  - Coordinate assignment to each dimension (for all of above descriptions see Tab. 5.1 at page 18)
  - Data type (see Tab. 5.2 at page 19).
  - Use simenv.chk to check variables description in model output description file <model>.mdf
- Addressing of model output data fields or parts of it is done in model output post-processing by corresponding model output variables names.

- For variables with a dimensionality greater than 0 it is possible to address only a part of the whole variable field by
  - Specifying for a dimension an index range by
    i = index_value_1 { : index_value_2 }
    index_value_1 ≤ index_value_2
    index_value_2 = index_value_1 if index_value_2 is missing.
    i= stands for index addressing
  - Specifying for a dimension an coordinate range by
    c = coordinate_value_1 { : coordinate_value_2 }
    coordinate_value_1 ≤ coordinate_value_2 for strictly increasing coordinate values
    coordinate_value_1 ≥ coordinate_value_2 for strictly decreasing coordinate values
    coordinate_value_1 = coordinate_value_2 if coordinate_value_2 is missing
    c= stands for coordinate addressing
  - Index and coordinate ranges are separated from each other by a comma, the sequence of ranges for all dimensions is enclosed in brackets and is appended after the variable name.
    For one variable c= and i= can be used in mixed mode for different dimensions.
    * denotes the complete range of a dimension.
    c= * is identical to i= * is identical to *
  - In the general SimEnv configuration file <model>.cfg (see chapter 10.1 at page 81) a global default for index and/or coordinate addressing is established for the whole post-processing session. This global default can be overwritten locally by using c= and/or i=.

Having a model variable definition as in Example 5.1 at page 20 then in model output post-processing

```
atmo                                    and
atmo(*,*,*,*)                           and
atmo(c=*,*,i=*,*)                       and
atmo(c=88:-88,c=-178:178,c=1:16,c=1:20) and
atmo(i=1:45,i=1:90,i=1:4,i=1:20)        and
atmo(i=1:45,c=-178:178,*,*)             and
atmo(1:45,1:90,1:4,1:20)                and (with address_default = index in model.cfg)
atmo(1:45,c=-178:178,1:4,1:20)          and (with address_default = index in model.cfg)
                                  all address all 45*90*4*20 values and
                                  the following holds true for this addressed variable:
                                  Dimensionality = 4
                                  Coordinates = lat , lon , level , time
                                  Extents = 45 , 90 , 4 , 20
atmo(*,*,*,c=11:20)               addresses all values of last 10 decades
                                  Dimensionality = 4
                                  Coordinates = lat , lon , level , time
                                  Extents = 45 , 90 , 4 , 10
atmo(*,*,c=1,c=1)                 addresses all values of the first decade for level 1
                                  Dimensionality = 2
                                  Coordinates = lat , lon
                                  Extents = 45 , 90
atmo(c=0,*,1,i=20)                addresses all values of level 1for the last decade at
                                  equator
                                  Dimensionality = 1
                                  Coordinates = lon
                                  Extents = 90
atmo(i=23,*,1,i=20)               addresses all values of level 1for the last decade at
                                  equator
                                  Dimensionality = 1
                                  Coordinates = lon
                                  Extents = 90
```

`atmo(c=0,c=2,c=1,c=20)`	addresses the value for the last decade at (lat,lon,level,time) = (0°,2°,1,20) Dimensionality = 0 Coordinates = (without) Extents = (without)
`atmo(c=0,c=1:9,c=1,c=20)`	addresses the values for the last decade at (lat,lon,level,time) = (0°,2°,1,20) and (0°,6°,1,20) Dimensionality = 1 Coordinates = lon Extents = 2
`atmo(c=0,c=1,c=1,c=20)`	error in addressing: c=1 for lon does not exist

Example file: world.post_bas

Example 8.1 *Addressing model output variables in model output post-processing*

8.3 Operators

- Operators transform dimensionality, dimensions, and coordinates of the their non-character operator arguments into unique dimensionality, dimensions and coordinates of the operator result.
  There are
  - Multi-argument operators that demand a certain relation between dimensionalities, dimensions and coordinates of their arguments
  - Single-argument operators that replicate dimensionality, dimensions and coordinates from the only argument to the operator result
  - Operators that increase dimensionality of the operator result and assign new coordinates to the additional dimensions (check Tab. 8.1 below) or form new coordinates from resulting target adjustments (see chapter 8.5.1 for behavioural analysis).
- SimEnv post-processing operators may have two special types of arguments:
  - Character arguments:
    Only character strings enclosed in ‘ ‘ are valid as arguments. Some built-in operators (e.g., count) have a pre-defined set of valid character argument strings (e.g., for operator count strings all, def, and undef)
  - Integer or float constant arguments:
    Only constants in appropriate format are valid as arguments. Model variables of dimensionality 0 or general operands with dimensionality 0 are invalid.
    Examples: `bios_g, sin(bios_g)`
  - If defined, character and constant arguments are always the first arguments of an operator. If both argument types are defined for an operator then the sequence is character arguments followed by constant arguments.
- Operators are generic with respect to the data types of their operands: Each non-character argument can be used with operands of all defined data types (see chapter 5.1). Internally, arguments of any type are converted to real*4 / float representation. This may lead to undefined real*8 arguments in real*4 representation.
- Results of SimEnv post-processing operators are always of the type real*4 / float.
- SimEnv post-processing follows the standard approach for description of operators for basic as well as advanced built-in or user-defined operators.
- Advanced built-in or user-defined operators
  - Have a unique name and a number of operands
  - The sequence of operands is enclosed in parentheses directly after the operator name
  - Operands are separated by a comma.
  - Recursions of the same operator (also for user-defined operators) are possible.
    Example: `log10(min_n(3 , min_n(log10(atmo(*,*,1,c=20)) , 400) , 10*bios_g))`
- Elemental operators use the common form of notation:
  Example: `atmo_g + 345`

Coordinate name	Coordinate values (check Tab. 11.4 for syntax)	Operators
bin	equidist_end 1(1) number_of_bins	hgr, hgr_l, hgr_e
index	equidist_end 1(1) operator_dependent	minprop, maxprop, minprop_l, maxprop_l
run	equidist_end 1(1) number_of_runs	ens
stat_measure	equidist_end 1(1)10	stat

Tab. 8.1 *Additional coordinates*

8.3.1 Operands and Coordinate Checking

The requirement for a lot of operators that their arguments must have same coordinates for same dimensions may restrict application of post-processing especially for hypothesis check heavily. To enable a broader flexibility with respect to this situation a general solution is provided by SimEnv post-processing: With the sub-keyword coord_check in the general configuration file <model>.cfg three different modi can be assigned globally to SimEnv post-processing:

- coord_check = strong
  To ensure for two arguments with same dimensionalities and extents to have same coordinates it is necessary that
  - Assigned coordinate values for corresponding dimensions are unique
  - Assigned coordinate names for corresponding dimensions are unique

  coord_check = strong is the default
- coord_check = weak
  To ensure for two arguments with same dimensionalities and extents to have same coordinates it is necessary that
  - Assigned coordinate values for corresponding dimensions are unique
  - Assigned coordinate names may differ.

  Coordinate description of an appropriate operator result dimension is delivered from the first operand.
- coord_check = without
  To ensure for two arguments with same dimensionalities and extents to have same coordinates
  - Neither coordinate names nor coordinate values for corresponding dimensions are checked

  Coordinate description of an appropriate operator result dimension is delivered from the first operand.

Check Example 8.2 for some instances.

Having a model variable definition as in Example 5.1 at page 20 then the checking rules for coordinates are applied in the following manner to operands with dimensionality 1:

Expression	Same coordinates for <coord_check> =		
	strong	weak	without
`bios(*,*,*) + atmo(c=84:-56,*,c=1,*)` (same coordinate names, same coordinate values)	yes	yes	yes
`atmo_g(*) + hgr(20,atmo)` (differing coordinate names, same coordinate values)	no	yes	yes
`atmo_g(c=10:20) + atmo_g(c=6:16)` (same coordinate names, differing coordinate values)	no	no	yes
`atmo_g(c=20) + atmo(c=0,c=2,c=1,c=1)` (two operands with dimensionality 0)	yes	yes	yes

While determination of coordinate information is unique for <coord_check> = strong,
coordinate information is delivered from the first summand for <coord_check> = [weak | without].

Example 8.2 *Checking rules for coordinates*

8.4 Built-in Elemental, Basic, and Advanced Operators

8.4.1 Built-in Elemental Operators

Name	Meaning	Argument restriction(s) / result description (see Tab. 8.3)	Argument value restriction	Precedence
(	left parenthesis	-		first
)	right parenthesis	-		first
arg1 ** arg2	exponentiation	(2)	arg1 > 0	second
arg1 * arg2	multiplication	(2)		third
arg1 / arg2	division	(2)	arg2 ≠ 0	third
arg1 + arg2	addition (dyadic +)	(2)		fourth
arg1 – arg2	subtraction (dyadic -)	(2)		fourth
+ arg	identity (monadic +)	(1)		fourth
– arg	negation (monadic -)	(1)		fourth

Tab. 8.2 *Built-in elemental operators*

- n-dimensional matrix algebra of built-in elemental operators is performed element by element
  Example:
  `atmo(*,*,1,*) * bios(*,*,*)` = "atmo(i,j,1,k) * bios(i,j,k)" for all addressed (i,j,k)
- If an argument value restriction is not fulfilled for an operand element the corresponding element of the operator result is undefined.

Argument restriction(s) / result description		Argument restriction(s)	Result description (check chapter 8.1 for syntax)
(1)		dimensionality, extents and coordinates of the only non-character argument arg can be arbitrary	same dimensionality, extents and coordinates as the only non-character argument: dim(res) = dim(arg) ext(res,j) = ext(arg,j) for all j coord(res,j) = coord(arg,j) for all j
(2) **=** (2.1) or (2.2)	(2.1)	all non-character arguments with same dimensionality, extents and coordinates **(*)** arg	same dimensionality, extents and coordinates as all the non-character arguments: dim(res) = dim(arg) ext(res,j) = ext(arg,j) for all j coord(res,j) = coord(arg,j) for all j
	(2.2)	some non-character arguments with same non-zero dimensionality, extents and coordinates **(*)** arg, all the other non-character arguments with dimensionality zero	same dimensionality, extents and coordinates as all the non-character arguments with non-zero dimensionality: dim(res) = dim(arg) ext(res,j) = ext(arg,j) for all j coord(res,j) = coord(arg,j) for all j the zero-dimensional argument is applied to each element of the non-zero dimensional argument

Argument restriction(s) / result description		Argument restriction(s)	Result description (check chapter 8.1 for syntax)
(3)		dimensionality, extents and coordinates of the only non-character argument can be arbitrary	dimensionality 0: dim(res) = 0
(4) = (4.1) or (4.2)	(4.1)	all non-character arguments with same dimensionality, extents and coordinates **(*)**	dimensionality 0: dim(res) = 0
	(4.2)	some non-character arguments with same non-zero dimensionality, extents and coordinates **(*)**, all the other non-character arguments with dimensionality zero	dimensionality 0: dim(res) = 0 the zero-dimensional argument is applied to each element of the non-zero dimensional argument
(5)		dimensionality, extents and coordinates of the first non-character argument arg can be arbitrary, all the other character arguments have to have dimensionalities, extents and coordinates **(*)** of this argument or have to have dimensionality 0	same dimensionality, extents and coordinates as the first non-character argument: dim(res) = dim(arg) ext(res,j) = ext(arg,j) for all j coord(res,j) = coord(arg,j) for all j
(6)		without arguments	dimensionality 0: dim(res) = 0

Tab. 8.3 *Classified argument restriction(s) / result description*
(*): for the different levels of checking a coordinate description see chapter 8.3.1

8.4.2 Built-in Basic and Advanced Operators

Name	Meaning	Argument restriction(s) / result description (see Tab. 8.3)	Argument value restriction	Example
Basic operators				
abs(arg)	absolute value	(1)		`abs(-3) = 3.`
dim(arg1,arg2)	positive difference	(2)		`dim(10,5) = 5.` `dim(5,10) = 0.`
exp(arg)	exponentiation	(1)		`exp(1.) = 2.7183`
int(arg)	truncation value	(1)		`int(7.6) = 7.` `int(-7.6) = -7`
log(arg)	natural logarithm	(1)	arg > 0	`log(2.7183) = 1.`
log10(arg)	decade logarithm	(1)	arg > 0	`log10(10) = 1.`
mod(arg1,arg2)	remainder	(2)	arg2 ≠ 0	`mod(10,4) = 2.`
nint(arg)	round value	(1)		`nint(7.6) = 8.`
sign(arg)	sign of value	(1)		`sign(-3) = -1.` `sign(0) = 0.`
sqrt(arg)	square root	(1)	arg ≥ 0	`sqrt(4) = 2.`
Trigonometric operators				
sin(arg)	sine	(1)		`sin(0) = 0.`
cos(arg)	cosine	(1)		`cos(0) = 1.`
tan(arg)	tangent	(1)	arg ≠ π/2±n*π	`tan(0) = 0.`
cot(arg)	cotangent	(1)	arg ≠ ±n*π	`cot(1.5708) = 0.`
asin(arg)	arc sine	(1)	abs(arg) ≤ 1	`asin(0) = 0.`

Name	Meaning	Argument restriction(s) / result description (see Tab. 8.3)	Argument value restriction	Example
acos(arg)	arc cosine	(1)	abs(arg) ≤ 1	`acos(1) = 0.`
atan(arg)	arc tangent	(1)		`atan(0) = 0.`
acot(arg)	arc cotangent	(1)		`acot(0) = 1.5708`
sinh(arg)	hyperbolic sine	(1)		`sinh(0) = 0.`
cosh(arg)	hyperbolic cosine	(1)		`cosh(0) = 1.`
tanh(arg)	hyperbolic tangent	(1)		`tanh(0) = 0.`
coth(arg)	hyperbolic cotangent	(1)	arg ≠ 0	`coth(3.1416) = 1.`
Miscellaneous operators				
classify(arg1, arg2)	classify arg2 into arg1 classes	(1) dim(arg2) > 0 arg1 = number of classes 2 ≤ arg1 ≤ number_of_values of arg2 = 0: automatic determination: number of classes = max(2,number_of_values/10) integer constant argument		`classify( (10,atmo)`
clip(arg1,arg2)	clip arg2 according to arg1	dim(arg2) > 0 dim(res), ext(res,i) depend on arg1 and arg2 arg1 = clip range character argument		`clip( '0,*,1,10', atmo)`
cumul(arg1,arg2)	cumulates arg2 according to arg1	(1) dim(arg2) > 0 arg1 = cumulation indicator per dimension character argument		`cumul('0001', atmo)`
experiment (arg1,arg2,arg3)	include an other experiment	(1) arg1 = experiment directory character argument arg2 = model experimented with character argument arg3 = result from this experiment		`experiment( 'mod_res','mod', avg(atmo)-400.)`
flip(arg1,arg2)	flips arg2 according to arg1	(1), but coordinates is also flipped dim(arg2) > 0 arg1 = flip indicator per dimension character argument		`flip('0001', atmo)`
if(arg1, arg2,arg3,arg4)	conditional if-construct	(5) arg1 = comparison operator character argument arg2 = comparator arg3, arg4 = new assignments		`if('<',atmo,400, atmo)`
mask (arg1,arg2,arg3)	masks values (set them undefined) by comparing arg2 and arg3 using operator arg1	(5) arg1 = comparison operator character argument		`mask('<',atmo, 400)`
matmul(arg1, arg2)	matrix multiplication	dim(arg1) = dim(arg2) = dim(res) = 2 ext(res,i) according to matrix multiplication rules		
nr_of_runs	number of single runs in the experiment	(6)		`nr_of_runs()`

Name	Meaning	Argument restriction(s) / result description (see Tab. 8.3)	Argument value restriction	Example
rank(arg1,arg2)	assigns rank numbers to arg2 according to ranking type argument arg1	(1) dim(arg2) > 0 arg1 = ranking type [tie_plain | tie_min | tie_avg]		`rank('tie_avg', atmo)`
run(arg1,arg2)	values of arg 2 for the selected single run number explicitly or implicitly coded in arg1	(1) arg1 = run number selection = 0 for default run (all experiment types) = <run_number> (for Monte-Carlo analysis, 0 ≤ arg1 ≤ number_of_runs) = filter argument (for behavioural analysis, same as filter argument of operator behav, check chapter 8.5.1) character argument		`run('0',atmo)` `run('sel_t(p1(4))',atmo)`
table_fct(arg1, arg2)	table function with linear interpolation of table arg1 applied to arg2	(1) arg1 = file name character argument		`table_fct ('table.usr', atmo)`
transpose(arg1, arg2)	transpose arg2 according to sequence in arg1	dim(arg2) > 1 dim(res) = dim(arg2) ext(res,i) = ext(arg2,j) (re-sorted) arg1 = transpose sequence character argument		`transpose ('3142',atmo)`
undef( )	undefined value	(6)		`undef()`

Tab. 8.4 *Built-in advanced operators (without standard aggregation / moments operators)*

The following explanations yield for the operators in Tab. 8.4:

- **All operators but experiment and matmul** are applied to each element of the argument(s). These operators deal with an unfulfilled argument value restriction for an operand element in a way that the corresponding element of the operator result will be undefined.
- The **operator classify** transforms an operand arg2 that has dimensionality > 0 into arg1 classes 1 ,..., arg1. Classes are assumed to be equidistant. Lower boundary of the dynamics range from arg2 assigned to class number 1 is the minimum of all values of arg2, upper boundary assigned to class number arg1 is the maximum of all values of arg2.
- The **operator clip** clips an operand that has dimensionality > 0. The portion to clip from the operand arg2 is described by character argument arg1. Argument arg1 uses syntax for model output variable addressing (see chapter 8.1 at page 51). Note, that for all dimensions of argument arg2 lower bound index is 1. This applies also to model variables where the lower bound index is unequal 1 in the model output description file. In general, extents differ between the result of the operator clip and the argument arg2. Clip reduces the dimensionality of the result with respect to the argument arg2 to clip if the portion to be clipped is limited to one value for at least one dimension.
  A character argument arg1 = '*,*,...' results for operator clip in the identity of argument arg2.
- The **operator cumul** cumulates an operand that has dimensionality > 0. Cumulation is performed for all values of the argument arg2 from the first addressed index position up to the current index position. With the character argument arg1 these dimensions are identified that are to be cumulated. Character 1 at position i means cumulation across dimension i while a 0 stands for no accumulation. cumul('0...0',arg) results in the identity to arg.
- The **operator experiment** is to access to external SimEnv model output from the same or an other model performed with the same or another experiment type and stored in the same or in an other model output format. Model variables can differ from that used for the current model. Use for the experiment directory arg1 always that working directory the external experiment was started from. The external ex-

periment is always post-processed completely over all single runs. Environment variables from operating system level in the specification of the directory are not allowed. If the imported expression has same coordinate names as defined in the original experiment coordinate descriptions are checked against each other, otherwise coordinate descriptions are imported from the external into the original experiment.

Attention: Make sure no SimEnv service is running from the experiment directory arg1.

- The **operator flip** enables flipping of variable fields. For a one-dimensional field (a vector) flip changes the value of the first index position with the value of the last position, the value of the second position with that of the last but one position, etc. With the character argument arg1 these dimensions are identified that are due to flip. Character 1 at position i means flipping also for dimension i while a 0 stands for no flipping at this dimension. Flipping includes adaptation of coordinates and the assigned grid. cumul('0...0',arg) results in the identity to arg.
- The **operator if** supplies a general conditional if-construct. It operates for each element of the operand arg2 in the following way:

```
if ( condition(arg1,arg2) ) then
        res=arg3
else
        res=arg4
endif
```

with condition(arg1,arg2):	arg2 < 0	(arg1 = '<')
	arg2 ≤ 0	(arg1 = '<=')
	arg2 > 0	(arg1 = '>')
	arg2 ≥ 0	(arg1 = '>=')
	arg2 = 0	(arg1 = '=')
	arg2 != 0	(arg1 = '!=')
	arg2 def	(arg1 = 'def')
	arg2 undef	(arg1 = 'undef')

- The **operator mask** supplies a method to mask values. It operates for each element of the operand arg2 in the following way:

```
if ( condition(arg1,arg2,arg3) ) then
        res=undef()
else
        res=arg2
endif
```

with condition(arg1,arg2,arg3):	arg2 < arg3	(arg1 = '<')
	arg2 ≤ arg3	(arg1 = '<=')
	arg2 > arg3	(arg1 = '>')
	arg2 ≥ arg3	(arg1 = '>=')
	arg2 = arg3	(arg1 = '=')
	arg2 != arg3	(arg1 = '!=')

- The **operator matmul** performs a simple matrix multiplication for 2-dimensional arguments arg1 and arg2.
- The **operator nr_of_runs** returns the number of performed single runs of the current post-processed experiment without the run number 0 of the nominal constellation. It does not have an argument.
- The **operator rank** transforms all values of an operand arg2 that has dimensionality > 0 into their ranks. Small values get low ranks, large values get high ranks. Character argument arg1 determines how to rank ties, i.e., values of arg2 that are identical or have a maximum absolute difference of 1.e-6:

Assume an argument arg2 with 6 values	(4., 2., 4., 4., 4., 8.).
arg1 = 'tie_plain' returns ranks	(2 , 1 , 2 , 2 , 2 , 3) same minimal rank 2; next rank is 3, does not take into account the number of identical values
arg1 = 'tie_min' returns ranks	(2 , 1 , 2 , 2 , 2 , 6) same minimal rank 2; next rank is 6, taking into account the number of identical values
arg1 = 'tie_max' returns ranks	(3.5 , 1 , 3.5 , 3.5 , 3.5 , 6) same average rank 3.5; next rank is 6, taking into account number of identical values

- The **operator run** selects a single run from the run ensemble. The operator run must not contain experiment-specific (multi-run) operators as operands, while these operators may refer to the operator run. Additionally, run must not contain itself as an argument.
  Character argument arg1 can hold explicitly the run number string Monte-Carlo analysis. Run number 0 corresponds with the default single run 0 and is permitted as arg1 for all experiment types. For behavioural analysis a filter of the operator behav (see chapter 8.5.1) is applied as arg1 to select a unique run number unequal zero. For this purpose, a single run can be selected by the select-operator (check Tab. 8.11) of the operator behav. For Monte-Carlo analysis, single runs with a run number unequal zero are selected explicitly. Therefore, the file <model>.edf_adj holds the targets to be adjusted to the default values for the current experiment. Run number n corresponds with record number n of this file. For more information on <model>.edf_adj check chapter 6.1 at page 33. For examples see Example 8.4 and Example 8.5.
- With the **operator table_fct** a table function arg1 is applied to each element of the operand arg2. If necessary, table values are interpolated linearly. Outside the definition range of the table function the first and/or the last table value is used. File arg1 to hold the table function must be an ASCII file with two columns: The first column of each line is the argument value x, the second column the function value f(x). Arguments have to be ordered in a strictly increasing manner. Syntax rules for comments and separators in the table function file are the same as for user defined files (check chapter 11.2). Environment variables from operating system level in the specification of the file name arg1 are not allowed. Check the table function world.dat_tab in the examples directory of $SE_HOME for more information.
- The **operator transpose** enables to transpose an operand that has a dimensionality > 1. Sequence of extents of the transposed result is described by character argument 1: It consists of figures 1 , ..., dim(arg2) where the figure sequence corresponds with the re-ordered sequence of the operator result extents.
  A character argument arg1 = '123...' results for operator transpose in the identity of argument arg2.
- The **operator undef** supplies a 0-dimensional result as undefined. This operator can be used in the if-operator.

Generic aggregation and moment operator	**Meaning**
min	minimum of values
max	maximum of values
sum	sum of values
avg	linear mean of values
var	variance of values
avgg	geometric mean of values
avgh	harmonic mean of values
avgw	weighted mean of values
hgr	histogram of values
count	number of values
minprop	minimal, suffix related property of values
maxprop	maximal, suffix related property of values

Tab. 8.5 *Built-in generic standard aggregation / moment operators*

The generic operators in Tab. 8.5 can be applied during model output post-processing to derive aggregations and moments from operands in different ways by appending suffixes to the generic operator name:

- Appending **no suffix**:
  Aggregate the only non-character argument(s)
  Result is a scalar (an operator result of dimensionality zero) for all but operators hgr, minprop and maxprop.
  For operator hgr dimensionality of the result is 1, the extent is the specified number of bins for the histogram and the coordinate assigned has the name bin. Coordinate values are equidistant with 1 as the first value and an increment of 1.

For operators minprop and maxprop dimensionality of the result is 1. For argument dimensionality greater / equal 1 extent of the result is equal to the argument dimensionality. Assigned coordinate name is index. Coordinate values are equidistant with 1 as the first value and an increment of 1. For argument dimensionality 0 result dimensionality is 0.

- Appending **suffix _n** (for n arguments)
  Aggregate an arbitrary number of arguments with argument restriction(s) / result description according to (2) in Tab. 8.3 at page 56 element by element
  Currently, only operators min_n and max_n are implemented.
  Result has same dimensionality, extents and coordinates as the arguments
- Appending **suffix _l** (for loop)
  Aggregate the only non-character argument(s) separately for selected dimensions. Dimensions to select are described by an additional loop character argument (corresponds with the group by-clause of the standard query language SQL of relational database management systems).
  Result has a lower dimensionality as the only non-character argument according to the loop character argument.
  For operator hgr_l, dimensionality is increased additionally by one, the additional extent is the specified number of bins for the histogram and the additional coordinate assigned to has the name bin. Coordinate values are equidistant with 1 as the first value and an increment of 1.

  For operators minprop_l and maxprop_l dimensionality is modified in the same manner like for operators minprop and maxprop, respectively.

Aggregation and moment operator	**Argument restriction(s) / result description (see Tab. 8.3)**
min(arg) max(arg) sum(arg) avg(arg) var(arg) avgg(arg) avgh(arg)	(3)
avgw(arg1,arg2)	(4.1) arg2 = weight
hgr(arg1,arg2)	dim(res) = dim(arg2)+1 ext(res,dim(res)) = number of bins coord(res,dim(res))= name = bin values = equidist_end 1(1) number of bins arg1 = number of bins: 4 ≤ arg1 ≤ number_of_values or = 0: automatic determination: number of bins = max(4,number_of_values/10) integer constant
count(arg1,arg2)	(3) arg1 = character argument = [all | def | undef]
minprop(arg) maxprop(arg)	dim(res) = 1 for dim(arg) > 1 ext(res,1) = dim(arg) dim(res) = 0 else returns the index of that element of arg where the extreme is reached the first time according to the processing sequence of the argument field arg by the Fortran column-wise storage model.

Tab. 8.6 *Built-in standard aggregation / moment operators without suffix*

Aggregation and moment operator	Argument restriction(s) / result description (see Tab. 8.3)
min_n(arg1,...,argn)	(4)
max_n(arg1,...,argn)	
minprop_n(arg1,...,argn)	(4)
maxprop_n(arg1,...,argn)	returns per result element the argument position (1 ... n) where the extreme is reached the first time. Processing sequence starts with arg1.

Tab. 8.7 *Built-in standard aggregation / moment operators with suffix _n*

Aggregation and moment operator	Argument restriction(s) / result description	
min_l(arg1,arg2) max_l(arg1,arg2) sum_l(arg1,arg2) avg_l(arg1,arg2) var_l(arg1,arg2) avgg_l(arg1,arg2) avgh_l(arg1,arg2) avgw_l(arg1,arg2,arg3)	dim(non-character argument(s)) > 1 ext(non-character argument(s)) = arbitrary dim(res), ext(res,i) according to arg1 and the non-character argument(s) arg1 = loop character argument	dim(arg2) = dim(arg3) ext(arg2,i) = ext(arg3,i) arg3 = weight
hgr_l(arg1,arg2,arg3)		dim(res) = 1 + dim(res) of all other operators ext(res,dim(res)) = number of bins coord(res,dim(res)) = name = bin values = equidist_end 1(1) number of bins arg2 number of bins 4 ≤ arg1 ≤ number_of_values or 0: automatic determination = max(4,number_of_values/10) integer constant
count_l(arg1,arg2,arg3)		arg2 = [all | def | undef] character argument
minprop_l(arg1,arg2) maxprop_l(arg1,arg2)	as above, but: dim(res) is increased by 1 w.r.t. above. ext(res,dim(res)) = dim(arg1) coord(res,dim(res)): name = index values = equidist_end 1(1)"n"	returns the indices of those elements of arg2 where the extreme is reached the first time according to arg1 and to a Fortran-like (column-wise) processing sequence of the argument field arg2.

Tab. 8.8 *Built-in standard aggregation / moment operators with suffix _l*

The loop character argument is characterised as follows:

- The length of the string is equal to the dimensionality of the non-character argument
- The string consists of 0 and 1
- 0 at position n means: aggregate over the corresponding dimension n of the argument
- 1 at position n means: do not aggregate over the corresponding dimension n of the argument
- Loop character arguments completely formed of 0 or 1 are forbidden

For the operator hgr_l bins are determined on the base of the minimum and maximum value of the total argument arg2.

Having a model variable definition as in Example 5.1 at page 20 and
assuming address_default=coordinate in <model>.cfg then in model output post-processing

`atmo_g`	value of variable atmo_g Dimensionality = 1 Coordinates = time Extents = 20
`if('<',atmo-10,10,atmo)`	maximum from atmo and 10 for each element of atmo equivalent with max_n(atmo,10) Dimensionality = 4 Coordinates = lat , lon , level , time Extents = 45 , 90 , 4 , 20
`clip('i=23,*,1,19:20',atmo)`	last two decades for level 1 at equator equivalent with atmo(i=23,*,1,19:20) Dimensionality = 2 Coordinates = lon , time Extents = 90 , 2
`avg(atmo(*,*,*,19:20))`	global all-level average over the last two decades Dimensionality = 0 Coordinates = (without) Extents = (without)
`maxprop(atmo)`	indices of this element of atmo where the maximum of atmo is reached the first time Dimensionality = 1 Coordinates = index Extents=4
`min_n(atmo(84:-56,*,1,19:20),10.)`	minimum per grid cell for level 1 without polar regions for the last two decades from atmo and 10 Dimensionality = 3 Coordinates = lat , lon , time Extents = 36 , 90 , 2
`min_l('10',atmo(20:-20,*,1,20))`	zonal tropical level-1 minima of atmo for the last decade Dimensionality = 1 Coordinates = lat Extents = 11
`minprop_l('10',atmo(20:-20,*,1,20))`	zonal tropical level-1 indices of those elements of atmo for the last decade where the minimum is reached the first time Dimensionality = 2 Coordinates = lat , index Extents = 11 , 2
`hgr_l('10',8,atmo(20:-20,*,1,20))`	zonal tropical level-1 histograms with 8 bins for the last decade Dimensionality = 2 Coordinates = lat , bin Extents = 11, 8
`avg_l('100',min_l('1011',atmo(20:-20,*,*,*)))`	temporally averaged all-level zonal tropical minima Dimensionality = 1 Coordinates = lat Extents = 11

```
table_fct('world.dat_tab',atmo)
                                Operator table_fct with table world.dat_tab applied to
                                each element of atmo
                                Dimensionality = 4
                                Coordinates = lat , lon , level , time
                                Extents = 45 , 90 , 4, 20
atmo - experiment('./other_dir','other_model',atmo)
                                Difference for atmo between the current experiment and
                                another model other_model, located in directory ./other_dir
                                Dimensionality = 4
                                Coordinates = lat , lon , level , time
                                Extents = according to definition of atmo in other_model

                                                                Example file: world.post_adv
```

Example 8.3 *Post-processing with advanced operators*

8.5 Experiment-Specific Operators

- Experiment-specific operators are to navigate and process in the experiment space.
- Experiment specific operators must not be applied recursively.
- Addressing a variable within an experiment specific operator normally results in application of the operator on the whole run ensemble or parts of it and in aggregating across the run ensemble according to the operator.
- Addressing a variable outside an experiment specific operator results in application of the basic, advanced and/or user-defined operator on the variable for the default run number 0 of the experiment.
- If the dimensionality of an operator result is higher than that of one of its operands the additional dimensions of the result are appended to the dimensions of the operand. Examples for such operators are ens (for Monte-Carlo analysis post-processing) and behav (for certain constellations of behavioural analysis post-processing).

Tab. 8.9 summarises multi-run standard aggregation / moment operators. They work on the whole run ensemble (for Monte-Carlo analysis) or parts of it (for certain constellations of behavioural analysis post-processing). They are used with suffix _e for Monte-Carlo analysis and without suffix for behavioural analysis.

Aggregation and moment operator	Argument restriction(s) / result description (see Tab. 8.3)
min(arg) max(arg) sum(arg) avg(arg) var(arg) avgg(arg) avgh(arg)	(1)
avgw(arg1,arg2)	(2.1) arg2 = weight
hgr(arg1,arg2) (heuristic probability density function)	dim(res) =dim(arg2)+1 ext(res,dim(res)) = number of bins coord(res,dim(res))= name = bin values = equidist_end 1(1) number of bins arg1 = number of bins 4 ≤ arg1 ≤ number_of_runs or 0: automatic determination = max(4,number_of_runs/10) integer constant
count(arg1,arg2)	(1) arg1 = [all | def | undef] character argument
minprop(arg) maxprop(arg)	(1) returns the run number where the extreme is reached the first time. Processing sequence starts with run number 1.

Tab. 8.9 *Multi-run standard aggregation / moment operators*

8.5.1 Behavioural Analysis

There is only one experiment specific operator for behavioural analysis. With this operator behav

- A single run can be selected from the run ensemble
- The complete run ensemble can be addressed
- Sub-spaces from the experiment space can be addressed and
- Sub-spaces can be projected by aggregation and moment operators

dependent on the way the experiment target space was to be scanned according to the comb-sub-keyword in the experiment description file.

To show the power of the operator behav the simple experiment layouts as described in Fig. 4.3 at page 12 are used as examples.

- With behav it is possible to address for any operand a single run out of the run ensemble by fixing values of experiment targets p1 and p2 (for Fig. 4.3 (a)), a value of the parallel targets p1 or p2 (for Fig. 4.3 (b)), and values of targets p3 and p1 or p2 (for Fig. 4.3 (c)). Dimensionality and extents of the operator result is the same as that of the operand.
- Without any selection in the target experiment space (p1,p2) and/or (p1,p2,p3) the dimensionality of the operator result is formed from the dimensionality of the operand enlarged by the dimensionality of the experiment space. Two additional dimensions are appended to the operand for Fig. 4.3 (a), one additional dimension for Fig. 4.3 (b), and two additional dimensions for Fig. 4.3 (c). For the latter two cases it is important which of the axis p1 and p2 is used for further processing and/or output of the operator result. The extents of the appended dimensions are determined by the number of target adjustments.
- As a third option it is possible to select only a sub-space out of the experiment space to append to the operand. For Fig. 4.3 (a) this could be the sub-space formed from the first until the third adjustment value of p1 and all adjustment values of p2 between 3 and 7. Dimensionality of the operator result increases by 2 and extents of these additional dimensions are 3 and 2 with respect to the corresponding Example 6.1 (a) in chapter 6.2.3 at page 35.

- The operator behav also enables to aggregate operands in the experiment space. In correspondence with the example in the last bullet point for Fig. 4.3 (a) the operand could be aggregated (e.g., averaged) over the first until the third adjustment value of p1 autonomously for all runs with different values of p2 and afterwards this intermediate result (that now depends only on p2) could be summed up for all adjustment values of p2 between 3 and 7. Consequently the result has the same dimensionality as the operand of behav. Sequence of performing aggregations is important.

Name	Meaning	Argument restriction(s) / result description	Argument value restriction
behav(arg1,arg2)	navigation in the experiment space for arg2 according to arg1	arg1= selection / aggregation filter character argument	

Tab. 8.10 *Experiment-specific operators for behavioural analysis*

Placeholder	Explanation
<filter>	' { <operator_1> {, <operator_2> ... {, <operator_n> } ... } } '
<operator>	[<select_operator> | <aggreg_operator> | <show_operator>]
<select_operator>	sel { _<target_value_type>} (<target_name> { <target_value_range> })
<aggreg_operator>	<aggreg_type> {_<target_value_type>} (<target_name> { <target_value_range> })
<show_operator>	show(<target_name>)
<target_name>	name of the experiment target according to the experiment description file
<target_value_type>	specification how to interpret <value_range> i as adjustment **i**ndices (indices always count from 1) v as adjustment **v**alues t as resulting **t**arget values
<target_value_range>	[(<value_1> { : <value_2> }) | (*)] for <value_2> = <nul> : value_2 = value_1 (*) : use all values from <target_name>
<aggreg_type>	an aggregation / moment operator from Tab. 8.5 at page 60. The following restrictions apply: • aggregations avgw and hgr can not be used • aggregation count has a differing syntax: count_<target_value_type> ([all | def | undef] , <target_name> { <target_value_range> })

Tab. 8.11 *Syntax of the filter argument 1 for operator behav*

The following rules yield for the operator **behav**:

- Generally, by the filter argument those runs from the run ensemble are selected and/or aggregated (here interpreted as filtered) that are used for the formation of the result.
  Consequently, if no filter is specified all runs are used.
  The select operator has to be specified only if values are to be restricted by a corresponding target value range.
  For the aggregation and the select operator the target value type is redundant if the value range represents the full range of values by <target_name> or <target_name>`(*)`.
  `sel(p1) = sel(p1(*)) = sel_i(p1) = sel_v(p1) = sel_t(p1)` and all are redundant.
- The show-operator can be used to force a certain experiment target to be used in the result of the operator behav if this target is used in parallel with other targets. By default, the first target of a parallel target sub-space as declared in the comb-line of the experiment description file is used in the behav-result.
- Aggregation operators reduce dimensionality of the covered experiment target space in the behav-result. The sequence of aggregation operators the first argument of the operator behav influences the result: Computation starts with the first aggregation operator and ends with the last:

`avg(p1), min(p2)` normally differs from `min(p2), avg(p1)`

- An unused experiment target in the selection and aggregation filter contributes with an additional dimension to the result of the operator behav. The extent of this additional dimension corresponds with the number of adjustments to this target in the experiment description file.
  A target that is restricted by any of the select operators also contributes with an additional dimension to the result of the operator behav if the number of selected values is greater than 1. The extent of the additional dimension corresponds with the number of selected values of this target by the select operator.
  Consequently, an empty character string arg1 forces to output the operand arg2 over the whole target space of the experiment.
- The name of the coordinate that is assigned to an additional dimension is the name of the corresponding target. Coordinate description and coordinate unit (see 5.1 at page 17) are associated with the corresponding information for the target from the experiment description file.
  Coordinate values are formed from resulting target values. For strictly ordered target adjustments in the experiment description file and finally for strictly ordered resulting target values the coordinate values are ordered accordingly in an increasing or decreasing manner. Unordered target adjustments and finally unordered target values are ordered in an increasing manner for coordinate usage.
  The result of the operator behav is always arranged according to ascending coordinate values for all additional dimensions.
- Independently from the sequence of the applied aggregation-, select- and show-operators the targets that contribute to additional dimensions of the result of the operator behav are appended to the dimensions of the operand arg2 of behav according to the sequence they are declared in the experiment description file (and **not** to the sequence they are used in the comb-line of the experiment description file).
  From parallel changing targets that target is used in this sequence that is addressed explicitly or implicitly by the show-operator.
- For experiment targets that are changed in the experiment in parallel, that increase dimensionality of the result and where a show-operator is missing the first target from this parallel sub-space in the comb-line is used in the result.
- For experiments that use an adjustment file (keyword file) instead of adjustment definitions (keyword comb) all experiment targets are assumed to be adjusted in parallel.

Having a model variable definition as in Example 5.1 at page 20 and
assuming address_default=coordinate in <model>.cfg
Assume the experiment layout in Example 6.1 (c) at page 36 and
the corresponding experiment description file (c) from Example 6.1 at page 35
then in result-processing

```
behav(' ',bios(*,*,20))
```
last time step of bios dependent on (p2,p1) and p3
Dimensionality = 4
Coordinates = lat , lon , p2, p3
Extents = 36 , 90 , 4 , 3

```
behav('show(p1)',bios(*,*,20))
```
last time step of bios dependent on (p1,p2) and p3
Dimensionality = 4
Coordinates = lat , lon , p1, p3
Extents = 36 , 90 , 4 , 3

```
behav('sel_t(p2(4)),sel_i(p3(1))',atmo(*,*,1,*))
```
select the single run out of the run ensemble for level 1
p2 = 4 and p3 = 3.3
Dimensionality = 3
Coordinates = lat , lon , time
Extents = 45 , 90 , 20

```
behav('sel_i(p2(1:3)),sel_v(p3(1:2))',atmo(*,*,1,20))
```
last time step of atmo for level 1 depend. on (p2,p1) and p3
use only runs for p2 = 1, 2, 3 and for p3 = 3.3, 4.5
Dimensionality = 4
Coordinates = lat , lon , p2, p3
Extents = 45 , 90 , 3 , 2

```
behav('avg_i(p2(1:3)),sel_i(p3(2:3))',atmo(*,*,1,*))
                                  average atmo for level 1 and for runs with p2 =1, 2, 3
                                  for each value of p3 = 4.5, 7.2
                                  Dimensionality = 4
                                  Coordinates = lat , lon , time , p3
                                  Extents = 45 , 90 , 20 , 2
behav('min(p2),max(p3)',avg(atmo(*,*,1,19:20)))
                                  determine single minima of avg(atmo) for level 1 and the
                                  last two decades for each value of p2
                                  afterwards determine from that the maximum over all p3.
                                  Dimensionality = 0
                                  Coordinates = (without)
                                  Extents = (without)
behav('max(p3),min(p2)',avg(atmo(*,*,1,19:20)))
                                  Result differs normally from min(p2),max(p3)
                                  (previous expression)
behav('count(def,p3),sel_i(p2=1)',bios(*,*,20))/3
                                  determine single numbers of defined values of
                                  bios for last decade for runs with p2=1.
                                  Result consists of values 0 (for water) and 1 (for land)
                                  Dimensionality = 2
                                  Coordinates = lat , lon
                                  Extents = 36 , 90
behav(' ',atmo(*,*,1,20)-run('sel_i(p1(1)),sel_i(p3(3))',
      atmo(*,*,1,20)))
                                  deviation of the last time step of atmo for level 1
                                  from the run with p1=1, p2=1, p3=3.3
                                  dependent on (p2,p1) and p3
                                  Dimensionality = 4
                                  Coordinates = lat , lon , p2, p3
                                  Extents = 45 , 90 , 4 , 3

                                                          Example file: world.post_c
```

Example 8.4 *Post-processing operator behav for behavioural analysis*

8.5.2 Monte-Carlo Analysis

Tab. 8.12 shows experiment specific operators for Monte-Carlo analysis that can be used in post-processing besides the general multi-run aggregation operators listed in Tab. 8.9 at page 65 and supplemented with a suffix _e. For a definition of these operators check Tab. 8.5 at page 60.

Name	Meaning	Argument restriction(s) / result description (see Tab. 8.3)	Argument value restriction
cnf(arg1,arg2)	positive distance of confidence measure from average avg_e(arg2)	(1) arg1 error probability	arg1 = [0.001 | 0.01 | 0.05 | 0.1] real*4 constant argument
cor(arg1,arg2)	correlation coefficient between arg1 and arg2	(2.1)	
cov(arg1,arg2)	covariance between arg1 and arg2	(2.1)	
ens(arg)	whole Monte-Carlo run ensemble	dim(res) = dim(arg)+1 ext(res,dim(res)) = number_of_runs coord(res,dim(res)) = name = run values = equidist_end 1(1) number_of_runs	
krt(arg)	kurtosis (4th moment)	(1)	
med(arg)	median	(1)	
qnt(arg1,arg2)	quantile of arg2	(1) arg1 quantile value	0. ≤ arg1 ≤ 100. real*4 constant argument
reg(arg1,arg2)	linear regression coefficient to forecast arg2 from arg1: arg2 = reg(arg1,arg2)*arg1 + n	(2.1)	
rng(arg)	range = max_e(arg) - min_e(arg)	(1)	
skw(arg)	skewness (3rd moment)	(1)	
stat(arg1,arg2, arg3,arg4, arg5)	basic statistical measures of arg5	dim(res) = dim(arg)+1 ext(res,dim(res)) = 10 coord(res,dim(res)) = name = stat_measure values = equidist_end 1(1)10	0. ≤ arg1 < arg2 ≤ 100. quantile values real*4 constant arguments arg3, arg4 = [0.001 | 0.01 | 0.05 | 0.1] arg3 < arg4 error probability for confidence distance measure real*4 constant arguments

Tab. 8.12 *Experiment-specific operators for Monte-Carlo analysis (without standard aggregation / moment operators)*

The following explanations yield for the operators in Tab. 8.12:

- The operator **stat** supplies basic statistical measures for argument arg5. The operator stat is a stand-alone operator: It must not be operand of any other operator. Contrary, argument arg5 can be composed from other non-multi-run operators. To store the statistical measures, dimensionality of stat is that of argument arg5, appended by an additional dimension with an extent of 10. Appended coordinate description is pre-defined by SimEnv (check Tab. 8.1).

  These ten data fields correspond with the following statistical measures:
  1. Deterministic run (run # 0)
  2. Run ensemble minimum
  3. Run ensemble maximum
  4. Run ensemble average

5. Run ensemble variance
6. Run ensemble median
7. Run ensemble quantile of quantile value arg1
8. Run ensemble quantile of quantile value arg2
9. Run ensemble positive distance of confidence measure from run ensemble average for value arg3
10. Run ensemble positive distance of confidence measure from run ensemble average for value arg3

For the definition of the statistical measures check the corresponding single operators in Tab. 8.9 and Tab. 8.12. Operator stat has been designed for application of an appropriate visualization technique in result evaluation in future.

Having a model variable definition as in Example 5.1 at page 20 and
assuming address_default=coordinate in <model>.cfg
Assume the Monte-Carlo experiment from Example 6.2 (e) at page 39
then in model output post-processing

`avg_e(p1*atmo(*,*,1,19:20))` global run ensemble average of p1*atmo for level 1 and the last two decades
Dimensionality = 3
Coordinates = lat , lon , time
Extents = 45 , 90 , 2

`avg(atmo(*,*,1,19:20))` global average of atmo for level 1 and the last two decades for run number 0
Dimensionality = 0
Coordinates = (without)
Extents = (without)

`ens(atmo(*,*,1,20)` run ensemble values of atmo for level 1 and the last decade
Dimensionality = 3
Coordinates = lat , lon , run
Extents = 45 , 90 , 250

`minprop_e(atmo(*,*,1,19:20))` run ensemble run number for level 1 and the last two decades
where the minimum of atmo is reached the first time
Dimensionality = 3
Coordinates = lat , lon , time
Extents = 45 , 90 , 2

`var_e(atmo(*,*,1,19:20))-atmo(*,*,1,19:20)`
anomaly for run ensemble variance from the nominal run for level 1 the last two decades
Dimensionality = 3
Coordinates = lat , lon , time
Extents = 45 , 90 , 2

`var_e(atmo(*,*,1,19:20)-run('0',atmo(*,*,1,19:20)))`
global run ensemble variance of the anomaly of atmo for level 1 and the last two decades.
Differs normally from the previous expression
Dimensionality 4
Coordinates = lat , lon , time
Extents = 45 , 90 , 4 , 20

`hgr_e(0,min_l('10',atmo(20:-20,*,1,20)))`
histogram with 25 bins for the zonal tropical minima for level 1 and the last decade
Dimensionality = 2
Coordinates = lat , bin
Extents = 11 , 25

```
stat(25,75,0.01,0.05, min_l('10',atmo(20:-20,*,1,20)))
                                  basic statistical measures for the zonal tropical minima
                                  of atmo for level 1 and the last decade
                                  Dimensionality = 2
                                  Coordinates = lat , stat_measure
                                  Extents = 11 , 10

                                                                  Example file: world.post_e
```

Example 8.5 *Post-processing operators for Monte-Carlo analysis*

8.6 User-Defined Operators

8.6.1 Declaration of User-Defined Operator Dynamics

- User-defined operators consist of a declarative and a computational part.
  - In the declarative part consistency of the non-character operands are checked and dimensionality, extents and coordinates of the result are defined.
  - In the computational part the result of the operator in dependency of the operands is computed.
- User-defined operators are specified in the ASCII operator description file <model>.odf. This file is used to check user-defined operators syntactically during result-post-processing.
- Check usr_opr_<opr>.f and apply the assigned operator <opr> for examples of user-defined operators.
- In SimEnv the declarative and computational part of an user-defined operator <opr> is hosted in a file usr_opr_<opr>.f. The assigned executable has the name <opr>.opr and has to be located in this directory that is stated in <model>.cfg as the hosting directory opr_directory for user-defined operators.
- Use the shell script operator_f.lnk <opr> to compile and link from usr_opr_<opr>.f an executable <opr>.opr that represents the user-defined operator <opr>.
- Use the simenv.chk to check user-defined operators
- Any user-defined operator can be transformed directly without changes to a built-in operator
- The functions to declare and compute user-defined operators listed below use a named common block simenv.

Function name	Function description	Inputs / outputs / function value	Inputs / outputs / function value description
Functions to host declarative and computational part in <model>.f			
icheck_user_ def_operator ( )	checks consis-tency of operator arguments and defines dimen-sionality and di-mensions of result	integer*4 icheck_user_ def_operator (function value)	return code = 0 ok ≠ 0 inconsistency between operands
icompute_user_ def_operator (result)	computes result of the operator in dependency on operands	real*4 result(1) (output)	result vector of the operator
		integer*4 icompute_user_ def_operator (function value)	return code = 0 ok ≠ 0 user-defined interrupt of calculation

Tab. 8.13 *Operator functions: Declarative and computational part*

Function name	Function description	Inputs / outputs / function value	Inputs / outputs / function value description
Functions to get and put structure information in declarative and computational part			
iget_char_arg (iarg, char)	gets string and length of the string of a character argument	integer*4 iarg (input)	argument number
		character*100 char (output)	string of the character argument
		integer*4 iget_char_arg (function value)	length of character argument
iget_dim_arg (iarg, iext)	iarg4 > 0: gets dimensionality and extents of an argument iarg4 = 0: gets dimensionality and extents of the result	integer*4 iarg (input)	argument number, 0 for result
		integer*4 iext(9) (output)	extents iext(1) ... iext(iget_dim_arg) of argument / result
		integer*4 iget_dim_arg (function value)	dimensionality of argument / result
iget_len_arg (iarg)	iarg4 > 0: gets length of an argument iarg4 = 0: gets length of result	integer*4 iarg (input)	argument number, 0 for result
		integer*4 iget_len_arg (function value)	length of argument / result
iget_nr_arg ( )	gets number of arguments of the current operator	integer*4 iget_nr_arg (function value)	number of arguments
iget_type_arg (iarg)	iarg4 > 0: gets data type of an argument iarg4 = 0: gets data type of result	integer*4 iarg (input)	argument number, 0 for result
		integer*4 iget_type_arg (function value)	type of argument / result = -1 byte = 4 float = -2 short = 8 double = -4 int
iget_co_chk_ modus ( )	gets level of coordinate check for arguments according to <model>.cfg	integer*4 iget_co_chk_ modus (function value)	level of coordinate check for arguments = 0 without = 1 weak = 2 strong
iget_co_arg (iarg, ico_blk, ico_beg)	gets coordinate block numbers and coordinate begin numbers of an argument	integer*4 iarg (input)	argument number
		integer*4 ico_blk(9) (output)	block number of the coordinate ico_blk(1) ... ico_blk(idimens)
		integer*4 ico_beg(9) (output)	begin numbers of the coordinate ico_beg(1) ... ico_beg(idimens)
		integer*4 iget_co_arg (function value)	return code = 0 ok

Function name	Function description	Inputs / outputs / function value	Inputs / outputs / function value description
iget_co_val (ico_blk, ico_pos, co_val)	gets coordinate value at a position from a coordinate	integer*4 ico_blk (input)	block number of the coordinate
		integer*4 ico_pos (input)	position of the value to get within all coordinate values
		real*4 co_val (output)	coordinate value
		integer*4 iget_co_arg (function value)	return code = 0 ok = 1 ico_pos out of range = 2 storage exceeded
ichk_2args (iarg1, iarg2)	checks two arguments on same dimensionality, extents and coordinates	integer*4 iarg1 (input)	argument number
		integer*4 iarg2 (input)	argument number
		integer*4 ichk_2args (function value)	return code = 0 ok = 1 differing dimensionalities = 2 differing extents = 3 differing coordinates according to <model>.cfg = 4 iarg1=iarg2
iput_struct_res (inplace, idimens [, iext, ico_blk, ico_beg])	puts potential in-place-storage, dimensionality, extents, coordinate bock and begin numbers of the result Currently, only coordinates from the arguments can be assigned to the result. **Apply only in the declarative part**.	integer*4 inplace (input)	potential inplace-indicator for result. result can be computed in-place with the following non-character arguments = -1 all = 0 none > 0 e.g. = 135 with args 1, 3 or 5
		integer*4 idimens (input)	dimensionality of the result
		integer*4 iext(9) (input)	only for idimens > 0: extents iext(1) ... iext(idimens) of the result
		integer*4 ico_blk(9) (input)	only for idimens > 0: coordinate block numbers ico_blk(1) ... ico_blk(idimens) of the result
		integer*4 ico_beg(9) (input)	only for idimens > 0: coordinate begin numbers in block ico_blk ico_beg(1) ... ico_beg(idimens) of the result
		integer*4 iput_dim_res (function value)	return code = 0 ok ≠ 0 inconsistency between operands

Tab. 8.14 *Operator functions to get and put structural information*

All of these functions return -999 as an error indicator if the argument iarg is undefined.

Function name	Function description	Inputs / outputs / function value	Inputs / outputs / function value description
Functions to get and check argument values and put results in computational part			
arg1 (index) ... arg9 (index)	gets value of a non-character argument with index index	integer*4 index (input)	vector index of an argument
		real*4 arg1 ... arg9 (function value)	value of an argument **arguments of any type are transferred to real*4 representation**
clip_undef (value)	**overflow:** checks a real*8 value on an undefined real*4 result **underflow:** sets a real*8 value to zero if appropriate	real*8 value (input)	value to be checked
		real*4 clip_undef (function value)	clipped value normally identified with a result res e.g., res(i)=clip_undef(value8)
is_undef (value)	checks whether value is undefined before processing it	real*4 value (input)	argument to be checked
		integer*4 is_undef (function value)	= 0 value is defined = 1 value is undefined
set_undef ( )	sets a result to be undefined	real*4 set_undef (function value)	normally identified with a result res e.g., res(i)=set_undef( )

Tab. 8.15 *Operator function to get / check / put arguments and results*

n-dimensional matrices are forwarded to user-defined operators as one-dimensional vectors, using the Fortran column-wise storage model:
Matrices are stored column-wise to the vector, starting with the highest dimension.

In Example 15.7 at page 121 implementation of the user-defined operator div is described in detail.

8.6.2 Operator Definition File <model>.odf

<model>.odf is an ASCII file that follows the coding rules in chapter 11 at page 91 with the keywords, names, sub-keywords, and info as in Tab. 8.16. <model>.odf describes the user-defined operators.

<keyword>	<name>	<sub-keyword>	Line type	Max. number of lines	<info>	Explanation
odf	<nil>	descr	o	any	<string>	general operator descriptions
operator	operator_ name	descr	o	1	<string>	operator description
		nr_args	m	1	<integer_value>	number of arguments defined for the operator operator_name 0< <integer_value> < 10
		nr_charargs	m	1	<integer_value>	from <integer_val>: number of character arguments defined for the operator operator_name 0 ≤ <integer_value> ≤ nr_args

Tab. 8.16 *Elements of an operator description file <model>.odf*

To Tab. 8.16 the following additional rules and explanations apply:

- For the description of **line type** check Tab. 11.3 at page 92.

```
odf                  descr         Operator description for the
odf                  descr         examples in the SimEnv User's Guide

operator   char_test   descr         test character arguments
operator   char_test   nr_args      3
operator   char_test   nr_char_args  2

operator   corr_coeff  descr         correlation coefficient
operator   corr_coeff  nr_args      2
operator   corr_coeff  nr_char_args  0

operator   div         descr         division
operator   div         nr_args      2
operator   div         nr_char_args  0

operator   simple_div  descr         division without special cases
operator   simple_div  nr_args      2
operator   simple_div  nr_char_args  0

operator   mat_mul     descr         matrix multiplication
operator   mat_mul     nr_args      2
operator   mat_mul     nr_char_args  0
```

Example files: world_[f | c | cpp | py | sh].odf

Example 8.6 *User-defined operator description file <model>.odf*

8.6.3 Handling Undefined Results

In user-defined operators

- Check always whether an argument value val is undefined by is_undef(val) before it is processed.
- Set a result to be undefined by the function set_undef()
  Check usr_opr_div.f for a detailed example
- If things go so wrong that processing of the whole expression has to be stopped alternatively it is possible to
  - Set all elements of the results to be undefined
  - Set icompute_user_def_operator ≠ 0 (otherwise set it always = 0)
  - In both cases processing of the next operators will be suppressed and consequently processing of the expression will be stopped
  - Check usr_opr_char_test.f for a detailed example

8.7 Undefined Results

- By performing operator chains and because of possibly unwritten model output during simulation parts of the intermediate and/or final result values can be undefined within the real*4 / float data representation.
- If an operand is completely undefined the computation of the result is stopped without evaluating the following operands and operators.
- For nodata value representation check Tab. 10.12.

8.8 Macro Definitions

- A macro in model output post-processing is an abbreviation for an expression, consisting of operator chains applied on operands.
- Generally, they are model related and they are defined by the user.
- Macros are identified in an post-processing expression by the suffix _m.
- A macro is plugged into an expression by putting it into parentheses during parsing:
  `equ_100yrs_m*test_mac_m`
  from Example 8.7 below is identical to
  `(avg(atmo(c=20:-20,*,c=1,c=11:20))-400)*(1+(2+3)*4)`
- Macros must not contain macros.
- Use simenv.chk to check macros. During the macro check validity of the following information is not checked:
  - Un-pre-defined character arguments of built-in operators (check Tab. 15.8)
  - Constant arguments of built-in operators (check Tab. 15.9)
  - Character arguments of user-defined operators
  - Operators with respect to dimensionality and dimensions of its operands

In SimEnv macros are hold in the file <model>.mac. <model>.mac is an ASCII file that follows the coding rules in chapter 11 at page 91 with the keywords, names, sub-keywords, and info as in Tab. 8.17. <model>.mac describes the user-defined macros.

<keyword>	<name>	<sub-keyword>	Line type	Max. num-ber of lines	<info>	Explanation
mac	<nil>	descr	o	any	<string>	general macro descriptions
macro	macro_ name	descr	o	1	<string>	macro description
		unit	m	1	<string>	unit of the value of the macro
		define	m	any	<string>	macro definition string macro definition can be ar-ranged at a series of define-lines

Tab. 8.17 *Elements of an macro description file <model>.mac*

To Tab. 8.17 the following additional rules and explanations apply:

- For the description of **line type** check Tab. 11.3 at page 92.

```
mac                    descr       Macro definitions for the
mac                    descr       examples in the SimEnv User's Guide

macro  equ_100yrs      descr       2nd century tropical level 1 average
macro  equ_100yrs      unit        without
macro  equ_100yrs      define      avg(atmo(c=20:-20,*,c=1,c=11:20))

macro  tst             descr       test macro
macro  tst             define      1+(2+3)*
macro  tst             define      4
```

Example files: world_[f | c | cpp | py | sh].mac

Example 8.7 *User-defined macro definition file <model>.mac*

8.9 Miscellaneous

- Continuation of expressions on a new input line after , + - * /
- White spaces are filtered out from the input string, also from character arguments

9 Visual Experiment Evaluation

Experiment evaluation is based on application of visualization techniques to the output data, computed during experiment post-processing and stored in NetCDF format. Currently, a preliminary version is implemented.

Analysis and evaluation of post-processed data selected and derived from large amount of relevant model output benefits from visualization techniques. Based on metadata information of the post-processed experiment type, the applied operator chain, and the dimensionalities of the post-processor output pre-formed visualization modules are evaluated by a suitability coefficient how they can map the data in an appropriate manner.
The visualization modules offer a high degree of user support and interactivity to cope with multi-dimensional data structures. They cover among others standard techniques such as isolines, isosurfaces, direct volume rendering and a 3D difference visualization techniques (for spatial and temporal data visualization). Furthermore, approaches to navigate intuitively through large multi-dimensional data sets have been applied, including details on demand, interactive filtering and animation. Using the OpenDX visualization platform techniques have been designed and implemented, suited in the context of analysis and evaluation of simulated multi-run output functions.

Currently, visual experiment evaluation is the only SimEnv service that comes with a graphical user interface. In this user interface a help-services is implemented that should be used to gather additional information on how to select post-processed results for visualization and on visualization techniques provided by SimEnv.

10 General Control, Services, User Files, and Settings

In a general configuration file <model>.cfg the user controls general settings for the simulation environment. Besides simulation performance and model output post-processing SimEnv supplies a set of auxiliary services to check status of the model, to dump model and post-processor output and files and to clean a model from output files. General settings reflect case sensitivity, nodata values and other information related to SimEnv.

10.1 General Configuration File <model>.cfg

In the ASCII file <model>.cfg general SimEnv control variables can be declared. <model>.mdf is an ASCII file that follows the coding rules in chapter 11 at page 91 with the keywords, names, sub-keywords, and info as in Tab. 10.1.

<keyword>	<name>	<sub-keyword>	Line type	Max. number of lines	<info>	Explanation
cfg	<nil>	descr	o	any	<string>	general configuration description
general	<nil>	message_level	o	1	[info | warning | error]	specifies which message types to show during simenv.chk and in <model>.mlog
model	<nil>	out_directory	o	1	<directory>	model output directory
		out_format	o	1	[netcdf | ieee]	model output format
		out_size_ threshold	o	1	<integer_value>	file size threshold in kBytes for lumped model output
		out_ieee_ blocksize	o	1	<integer_value>	block size in kBytes for IEEE model output
experiment	<nil>	restart_ini	o	1	[no | yes]	perform <model>.ini for experiment re-start
		begin_run	o	1	<integer_value>	begin single run number
		end_run	o	1	[last | <integer_value>]	end single run number
		email	o	1	<string>	email notification address
postproc	<nil>	out_directory	o	1	<directory>	post-processing output directory
		out_format	o	1	[netcdf | ieee | ascii]	post-processing output format
		address_default	o	1	[coordinate | index]	post-processing address default for model variables
		coord_check	o	1	[strong | weak | without]	post-processing coordinate check by operators
		opr_directory	o	1	<directory>	directory the post-processors looks for user-defined operator executables

Tab. 10.1 *Elements of a general configuration file <model>.cfg*

To Tab. 10.1 the following additional rules and explanations apply:

- For the description of **line type** check Tab. 11.3 at page 92.

<keyword>	<sub-keyword>	<info>-default value (*)	For more information see
cfg	descr	<nil>	above
general	message_level	info	above
model	out_directory	./	above
	out_format	NetCDF	chapter 12
	out_size_threshold	10	below
	out_ieee_blocksize	50	below
experiment	restart_ini	no	chapter 7.2
	begin_run	0	chapter 7.1 - 7.3
	end_run	last	chapter 7.1 - 7.3
	email	<nil>	chapter 7.1
postproc	out_directory	./	above
	out_format	NetCDF	chapter 12
	address_default	coordinate	chapter 8.2 and below
	coord_check	strong	chapter 8.3.1 and below
	opr_directory	./	chapter 8.6

Tab. 10.2 *<info> value defaults for the general configuration file*
(*): in the case of absence of the appropriate sub-keyword

The following explanations yield:

- <string>, <directory>, and <integer_value> are placeholder for corresponding strings.
- **For <keyword> = general, <sub-keyword>= message_level:**
  Message output during simenv.chk and to <model>.mlog is controlled by this information.
  Specify info for errors, warnings and additional information
  warning errors and warnings
  error errors
  output.
- **For <keyword> = experiment, <sub-keyword>= [begin_run | end_run]:**
  SimEnv enables to perform an experiment partially by performing only an experiment slice out of the whole run ensemble (see chapter 7.3 at page 46). Therefor assign appropriate run numbers to this two descriptors. Make sure that begin and end run represent run number from the experiment (including run number 0) and that begin run ≤ end run. The string 'last' always represents the last simulation run of the whole run ensemble.
- **For <keyword> = experiment, <sub-keyword>= email:**
  After performing an experiment an email is sent to the email address specified in <string>.
- **For <keyword> = model, <sub-keyword> = out_size_threshold:**
  Specify here the threshold in kBytes for the sum of the size of all model output variables (according to their extents and data types) that is used to decide whether the SimEnv model output data for the whole run ensemble is stored into one file <model>.outall.[nc | ieee] or in single output files <model>.out<run_char>.[nc | ieee].
- **For <keyword> = model, <sub-keyword> = out_ieee_blocksize:**
  IEEE compliant model output for single files is written in single records with a length of <out_ieee_blocksize> kBytes. If <out_size_threshold> is less than this value, this value is adapted to <out_size_threshold>.
- **For <keyword> = postproc, <sub-keyword> = address_default:**
  During post-processing portions of multi-dimensional model output variables can be addressed by coordinate (c= ...) or index (i= ...) reference. A default is established here.
- **For <keyword> = postproc, <sub-keyword> = coord_check:**
  During post-processing feasibility of application of an operator on its operands is checked with respect to the coordinate description of the operands. Different levels of this check are possible. A default is established here.

Please keep in mind to ensure consistency of control settings in <model>.cfg across different SimEnv tasks. As an example you have to run experimentation, post-processing and dump with the same model output file size threshold out_size_threshold from <model>.cfg for binary output.

```
cfg           descr                 General configuration file for the
cfg           descr                 examples in the SimEnv User's Guide

general       message_level         info

model         out_directory         mod_out
model         out_format            netcdf
model         out_size_threshold    100

experiment    begin_run             0
experiment    end_run               last

postproc      out_directory         res_out
postproc      out_format            netcdf
postproc      address_default       index
postproc      coord_check           strong
postproc      opr_directory         ./
```

Example 10.1 *User-defined general configuration file <model>.cfg*

10.2 Main and Auxiliary Services

The following SimEnv service commands are available from the SimEnv home directory $SE_HOME. Besides experiment performance and model output post-processing there are additional auxiliary SimEnv services to check input information consistency, to monitor the status of simulation experiments, to dump files of model and post-processor output and to wrap up the SimEnv workspace.

SimEnv command	Use to
Main Services	
simenv.run <model>	prepare and **run** an experiment
simenv.rst <model>	**restart** an experiment
simenv.res <model> { [new | append | replace] } {<run>}	perform experiment **result** post-processing for run number <run> or for the whole run ensemble (<run> = -1, default). If post-processor output is written in NetCDF format afterwards simenv.vis can be started for this post-processor output file. Before entering post-processing these output files <model>.res<res_char>.[nc | ieee | ascii] and <model>.inf<res_char>.[ieee | ascii] with the highest two-digit number <res_char> are identified and new result files for <res+1> are written / the results are appended / or the result files are replaced by a new ones.
simenv.vis <model> { [latest | <res>] }	perform **visual** post-processor output visualization for that NetCDF post-processor output file with the highest two digit number <res_char> (latest, default) or with the file number <res>. Visualization runs on a remote host.
simenv.cpl <model> { <run> } { <file> }	**complete** sequence of SimEnv commands simenv.chk, simenv.run, simenv.res, simenv.vis simenv.res is performed with input file <file> (if available) and interactively, for both optionally only for single run <run>.

SimEnv command	Use to
Auxiliary Services	
simenv.chk <model>	**check** on model script files (<model>.run, <model>.rst, <model>.ini, <model>.end) check <model>.cfg <model>.odf <model>.mdf <model>.edf <model>.gdf <model>.mac existing model and post-processor output files generate pre-experiment output statistics
simenv.sts <model> { <sleep> }	get **status** of an experiment that was started from a login node of a parallel machine and that is running in a parallel or sequential job class of this machine. To use this command login to the parallel machine and change to the working directory the experiment was started from. Does only work properly for model start, but not for model restart.
simenv.dmp <model>	**dump** SimEnv model output and post-processor output files Files to dump have to match the SimEnv file name convention for model and/or post-processor output and are expected to be in the directories as stated in <model>.cfg. Model output variables and post-processor results in IEEE and/or ASCII format with a dimensionality greater than 1 are listed according to Fortran column-wise storage model for multi-dimensional fields. To use this command change to the working directory the experiment and post-processing were started from.
simenv.cln <model>	**clean** up model and post-processor output files Deletes all model output files, post-processor output files, log-files, and auxiliary files of a model. To use this command change to the working directory the experiment and post-processing were started from.
simenv.cpy <model>	**copy** all SimEnv example files <model>* from the examples directory of $SE_HOME to the current directory. Additionally, example files of user-defined operators and for models world_[f | c| cpp | py | sh]* common user defined files are copied. All files are only copied of they do not already exist in the current directory, this SimEnv service is started from.
simenv.hlp <topics>	acquire basic SimEnv **help** information for the specifies topics
simenv.key <user_name>	generate a ssh2-**key** to get password-free access to the visualization server. Start this service at machine aix02 only one time before the first use of simenv.vis. To get this access finally contact the SimEnv developers after running the service

Tab. 10.3 *Service commands*

Do not start a SimEnv service from a working directory, excepted simenv.sts, if there is a running SimEnv service that was started from this working directory.

10.3 User Scripts and Files

Script / file (in the current working directory)	Explanation	Exist status	For more information see chapter
<model>.cfg	ASCII user-defined general configuration file	optional	10.1
<model>.mdf	ASCII user-defined model (variables) description file	mandatory	5.1
<model>.edf	ASCII user-defined experiment description file	mandatory	6.1
<model>.mac	ASCII user-defined macro description file	optional	8.8
<model>.odf	ASCII user-defined operator description file	optional	8.6.2
<model>.gdf	ASCII user-defined GAMS model output description file	mandatory for GAMS models	5.6.2
<model>.run **(*)**	model shell script to wrap the model executable	mandatory	7.5
<model>.rst **(*)**	model shell script to prepare single model run restart	optional	7.5
<model>.ini **(*)**	model shell script to prepare simulation experiment additionally to standard SimEnv preparation	optional, mandatory for Python and GAMS models	7.5
<model>.end **(*)**	model shell script to clean up simulation experiment	optional, mandatory for GAMS models	7.5
<model>.<run_char>.err	touch this file in the model, in <model>.run and/or <model>.rst as an indicator to stop the complete experiment after <model>.run has been finished for single model run <run_char>	optional	7.5
simenv.jcf_par	user-specific job control file to submit a job by the LoadLeveler to a parallel class	optional	7.5
simenv.jcf_seq	user-specific job control file to submit a job by the LoadLeveler to a sequential class	optional	7.5
<opr>.opr (in the opr-directory according to <model>.cfg)	executable for user-defined operator <opr>	optional	8.6

Tab. 10.4 *User scripts and files*
***(*)**: make sure the shell script has execute permission by chmod u+x*

File	Generated in	Explanation
Permanent files		
<model>.edf_adj in the current working directory $SE_WD	experiment preparation	ASCII adjustment input file for the run ensemble derived from <model>.edf. Record no. n corresponds to single run no. n. Value no. m of each record is the adjustment for experiment target no. m in the edf-file
<model>.out<run_char> .[nc | ieee] in the model out_directory according to <model>.cfg	experiment performance if model output of a single run ≥ out_size_threshold from <model>.cfg	model output of run number <run> to be processed by the post-processor (for experiment performance in a parallel job class at a parallel machine files <model>.out<run_char>.[nc | ieee] are created temporarily)
<model>.outall .[nc | ieee] in the model out_directory according to <model>.cfg	experiment performance if model output of a single run < out_size_threshold from <model>.cfg	model output of all runs to be processed by the post-processor
<model>.elog in the current working directory $SE_WD	experiment performance	ASCII minutes file of experiment performance (simenv.run and all simenv.rst)
<model>.mlog in the current working directory $SE_WD	experiment performance	ASCII minutes file of model performance (simenv.run and all simenv.rst)
<model>.res<res_char> .[nc | ieee | ascii] in the experiment out_directory according to <model>.cfg	experiment post-processing	output file of a post-processor session
<model>.inf<res_char> .[ieee | ascii] in the experiment out_directory according to <model>.cfg	experiment post-processing	output structure description file of a post-processor session
run<run_char> sub-directory in the current working directory $SE_WD	experiment performance	only for GAMS models: Sub-directories of GAMS model performance that are kept according to the keep_runs sub-keyword in <model>.gdf
Temporary files (do not delete during experiment performance)		
simenv.cfg in the current working directory $SE_WD	all tasks	structured ASCII representation of <model>.cfg
<model>. [mdf | edf | odf | mac] _bin in the current working directory $SE_WD	experiment preparation, experiment post-processing	structured binary representation of <model>.[mdf | edf | odf | mac]
<model>.parid in the current working directory $SE_WD	experiment performance	ASCII file with job number from an experiment submitted by LoadLeveler for performance of the command simenv.sts
simenv_*.tmp in the current working directory $SE_WD	different tasks	auxiliary files
check chapter 5.6.3	experiment performance	auxiliary files and sub-directories for GAMS models

Tab. 10.5 *User files generated during SimEnv performance*

Fig. 10.1 sketches usage of SimEnv user scripts and files in the course of model interfacing, experiment preparation and performance, post-processing, and evaluation.

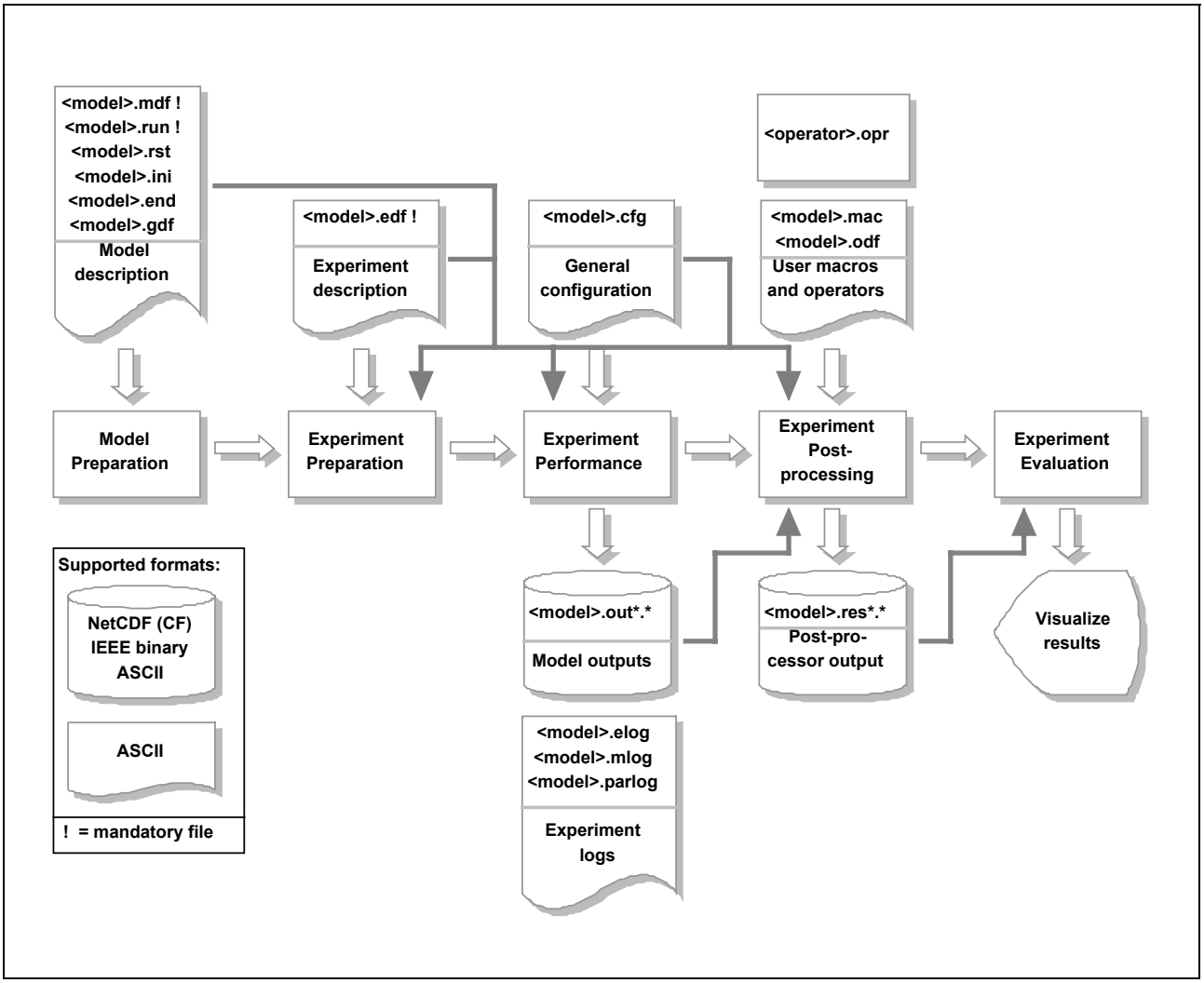


Fig. 10.1 *SimEnv user scripts and files*

10.4 Environment Variables

The following operating system environment variables are used by SimEnv:

Environment variable	Meaning	Explanation
SE_HOME	SimEnv home directory	has to be defined by the user Value = [/usr/local/simenv/1.03/bin | any previous SimEnv version at /usr/local/simenv/version_archive] Has to be included in the file $HOME/.profile for experiment submission to the LoadLeveler.
SE_RUN	run number of a single run	defined automatically within in <model>.run Value = <run_int>
SE_1STRUN	first single run of an experiment	defined automatically in <model>.run Value = [yes | no]
SE_WD	current SimEnv working directory	defined automatically within any SimEnv service Value = <path>
PYTHONPATH (only for interfacing Python and GAMS models)	path to search Python and Python files	has to be defined by the user. Value = machine dependent and has to be expanded by $SE_HOME Has to be included in the file $HOME/.profile for experiment submission to the LoadLeveler.

Tab. 10.6 *Environment variables*

10.5 Case Sensitivity

Where?	Entity	Sensitivity	Example
user-defined files (see Tab. 11.1)	• keyword • name exception: GAMS model file name in <model>.gdf • sub-keyword	case insensitive	`experiment  END_RUN last`

	• information <info>
	exceptions:
	• any file name:
	for <sub-keyword> = <string>_directory
	and for <info> starting with string “file”
	• <info> for <sub-keyword> = [descr | unit])
		case insensitive	`experiment  end_run LAST`
			`cfg  descr  This is...`
			`specific  comb  file AbC.d`

model interface	• variable and target name	case insensitive	`call simenv_put_f('ATMO',atmo)`
			
			`target_name='P1'`
			`target_value=1.`

Where?	Entity	Sensitivity	Example
			`. $SE_HOME/simnev_get_sh`
post-processing	• variable and target name • operator name • number • macro name • macro identifier _m	case insensitive	`exp(atmo) + 3*EXP(ATMO)`
	• character arguments of built-in operators with pre-defined values (check Tab. 15.8)	case insensitive	`count('ALL' , atmo)`
	• character arguments of built-in operators without pre-defined values	check Tab. 15.8	`table_fct('MyFile.dat' , atmo)` `experiment('../' , 'Model_f' , atmo)`
	• character arguments of user-defined operators	case sensitive	`char_test('arg11' , 'Arg21' , atmo)`

Tab. 10.7 *Case sensitivity of SimEnv entities*

10.6 Built-in Items, Reserved Names

Tab. 10.8 lists the built-in (pre-defined) model variables that are generally output during experiment performance to SimEnv model output structures and are available in model output post-processing without defining them in the model output description file <model>.mdf.

Model variable name	Dimensionality	Extents	Data type	Meaning
sim_time	0		float	elapsed simulation time in seconds (rounded to 2 decimal places) per single run for <model>.run

Tab. 10.8 *Built-in model variables*

Tab. 10.9 lists the built-in (pre-defined) coordinates that are used in model output post-processing when additional dimensions are generated by an operator.

Coordinate name	Operator	Meaning
bin	hgr, hgr_l, hgr_e	bin number
index	minprop, maxprop, minprop_l, maxprop_l	index number
run	ens	run number
stat_measure	stat	basic statistical measures
<target_name>	behav	target

Tab. 10.9 *Built-in coordinates*

Tab. 10.10 lists the built-in (pre-defined) shell script variables that are used in $SE_HOME/simenv_*_sh and finally in <model>.run.

Shell script variable name	Meaning
run_int	current run number as integer
run_char	current run number as character string
target_name	target name for simenv_get_sh
target_def_val	default target value for simenv_get_sh
simenv_hlp*_sh	auxiliary variable

Tab. 10.10 *Built-in shell script variables in $SE_HOME/simenv_*_sh*

Tab. 10.11 lists the reserved (forbidden) names and file names that can not be declared in user-defined files.

Element	Reserved (forbidden) names
model name <model>	simenv in any combination of upper and lower cases
<name> in user-defined files model.[mdf | edf | odf | mac] excepted for GAMS model source code file names (check chapter 11)	built-in model variables according to Tab. 10.8
	built-in coordinates according to Tab. 10.9
	built-in shell script variables according to Tab. 10.10
	special keywords in <model>.edf for behavioural analysis: [default | file]
<file_name> in <info> in user-defined files model.[mdf | edf | odf | mac] (check chapter 11)	SimEnv file names according to Tab. 10.4 and Tab. 10.5

Tab. 10.11 *Reserved names and file names in user-defined files and for models*

10.7 Nodata Representation

For model output with the SimEnv model coupling interface functions and for post-processor output the following data type specific nodata values are used to represent undefined (unwritten) model output or undefined post-processor output:

Data type	Nodata value
integer*1	127
integer*2	32767
integer*4	2147483648
real*4	3.4E+38
real*8	1.79D+308

Tab. 10.12 *Data type related nodata values*

11 Structure of User-Defined Files

Basic information to describe general control settings of SimEnv, model output variables, the experiment itself, macros and user-defined operators as well as GAMS model specific information is stored in user-defined files. They are ASCII files and have a common structure that is described in this chapter.

11.1 General Structure

All user-defined files listed in Tab. 11.1 have the same structure. They are ASCII-files with the following record structure:

{ <sep> } <keyword> <sep> { <name> <sep> } <sub-keyword> <sep> <info> { <sep> }

with

- <name> is the name of a
  - model variable
  - GAMS model source file
  - experiment target
  - coordinate
  - user-defined operator or
  - macro

  Declaration of <name> depends on the related keyword <keyword>
  <name> is case insensitive, excepted for the GAMS model source file
- <keyword> is a string
  Normally, more than one line with differing sub-keywords belong to one "keyword-block".
  <keyword> is case insensitive
- <sub-keyword> is a string
  Sub-keywords are defined only in relation to the user file and the keyword under consideration.
  <sub-keyword> is case insensitive
- <info> = <substring> { <sep> <substring> ... }
  is a string with user file, keyword and sub-keyword related information.
  <info> is case insensitive with the exception of any file name and/or directory and information for sub-keywords = [descr | unit]
- <sep> is a sequence of white spaces

Lines consisting only from separator characters as well as lines starting with a # as the first non-separator character are handled as comment lines. For case sensitivity of all information <info> in user files check Tab. 10.7 at page 89.

File	Contents	See description in chapter	See description at page
<model>.cfg	general configuration file	10.1	81
<model>.mdf	model output description file	5.1	17
<model>.gdf	GAMS description file	5.6.2	29
<model>.edf	experiment description file	6.1	33
<model>.odf	operator description file	8.6.2	75
<model>.mac	macro description file	8.8	76

Tab. 11.1 *User-defined files*

Element	Constraints
line length	max. 160 characters
<name>	max. 20 characters
	first character has to be a letter **(*)**
	must not end on _m **(*)**
	must not contain elemental operators and characters . and : **(*)** (check Tab. 8.2 at page 55)
	for further constraints check Tab. 10.11 at page 90
<info>	for sub-keyword = descr without <name>: max. 512 characters (total sum over all lines)
	for sub-keyword = descr with <name>: max. 128 characters
	for sub-keyword = <string>_directory: max. 70 characters must not contain environmental variables
	for sub-keyword = unit: max. 32 characters
	for further constraints check Tab. 10.11 at page 90

Tab. 11.2 *Constraints in user-defined files*
(*): excepted for GAMS model source code file names

The **line type** in a description table for a user-defined file specifies whether a keyword / sub-keyword combination can be omitted.

Abbreviation	User file	Explanation
m	all files	**m**andatory
o	all files	**o**ptional
c1	<model>.mdf keyword = variable sub-keyword = [coords | var_extents]	**c**onditional 1: forbidden for variables with dimensionality = 0 mandatory for variables with dimensionality > 0
c2	<model>.mdf keyword = variable sub-keyword = coord_extents	**c**onditional 2: forbidden for variables with dimensionality = 0 optional for variables with dimensionality > 0
c3	<model>.edf keyword = target sub-keyword = adjusts	**c**onditional 3: mandatory for experiment type = Monte-Carlo analysis forbidden for experiment type = local sensitivity analysis conditional for experiment type = behavioural analysis
c4	<model>.edf keyword = edf sub-keyword = Monte-Carlo keyword = target sub-keyword = sampling	**c**onditional 4: mandatory for adjusts = from specified distribution forbidden for adjusts = from external file
a	<model>.edf for behavioural analysis keyword = target sub-keyword = adjusts	**a**lternatively: either mandatory for all experiment targets or forbidden for all experiment targets
f	<model>.edf for local sensitivity analysis keyword = target sub-keyword = adjusts	**f**orbidden

Tab. 11.3 *Line types in user-defined files*

```
mac                 descr     This is a macro description file
mac                 descr     for the SimEnv User's Guide

macro    pol_atmo   descr     atmo outside polar reg., final time, level 1
macro    pol_atmo   unit      without
macro    pol atmo   define    atmo(c=84:-56,*,c=1,c=20)

macro    m1         define    avg(atmo_g(c=11:20))
...
```

Example 11.1 *Structure of a user-defined file*

Sequence of keyword and sub-keyword lines can be arbitrary. For reasons of readability it is recommended to use a block structure like in the above example. Sequence of names in the separated name spaces (name spaces of coordinates, model variables, experiment targets, user-defined operators, macros) during processing is determined by the sequence the name occur the first time in the appropriate user file.

11.2 Value Lists

For variables, coordinates and experiment targets value lists are supplied by the <info>-item. Value lists describe a sequence of values together with an order. The number of described values is greater than 1. Value lists may be restricted to strictly monotonic sequences. They follow the syntax rules in Tab. 11.4.

Value-list type	Syntax	Explanation
explicit	list <value$_1$> , ... , <value$_n$>	explicit list of values same syntax rules as for one record of a file with a value list (see below)
by reference	file {<path>/}<file_name>	file {<path>/}<file_name> contains the explicit value list
implicit with end-element	equidist_end <beg_val> (<incr_val>) <end_val>	description of an equidistant list of values with begin value <beg_val> increment <incr_val> end value <end_val> <beg_val> ≠ <end_val> <incr_val> ≠ 0.
implicit with number of values	equidist_nmb <beg_val> (<incr_val>) <nmb_vals>	description of an equidistant list of values with begin value <beg_val> increment <incr_val> number of values <nmb_vals> <beg_val> ≠ <end_val> <incr_val> ≠ 0. <nmb_vals> > 0, integer

Tab. 11.4 *Syntax rules for value lists*

Syntax rules for a file {<path>/}<file_name> with a list of values

- <path> must not contain environment variables from operating system level
- If <path> is specified in a relative manner it relates to the current working directory, the / a SimEnv service where <path> is used was started from.
- Has to be an ASCII file
- May be a multi-record file
- Max. record length: 1000 characters

- Values are separated from each other by white spaces or comma
- A series of connected (running) separators is treated as a single separator
- Record end is handled as a separator
- Real values can be stated in integer, real or exponential (scientific) format
- Records formed only from white spaces or records starting with first non-white space character # are handled as comments

1.	`list 3, 5, 7, 9, 11`	describes the five values 3, 5, 7, 9, 11
2.	`equisist_end 3 (2) 11`	is equivalent to 1.
3.	`equidist_nmb 3 (2) 5`	is equivalent to 1.
4.	`file my_vals.dat`	is equivalent to 1. with `my_vals.dat =  3, , 5, 7 9,11`
5.	`equidist_end 3 (2) 11.9`	is equivalent to 1.
6.	`equidist_end 11 (-2) 3`	differs from 1. – 4.: values are identical, ordering sequence differs

Example 11.2 *Examples of value lists*

12 Model and Post-Processor Output Data Structures

This chapter summarizes information on available data structures for model and post-processor output. SimEnv supports several output formats from the experiment and the post-processor. NetCDF is a self-describing data format and can be used for model and post-processor output. Another format specifications for both outputs is IEEE compliant binary format and ASCII for post-processor output. This chapter describes all the used data structures.

Dependent on the specification of the supported post-processor output formats in <model>.cfg model output can be stored in NetCDF format and post-processor output in NetCDF, IEEE or ASCII format.
During experiment performance model output is written either to single output files <model>.out<run_char>.[nc | ascii] per experiment single run or to a common output file <model>.outall.[nc | ieee] for all single runs from the experiment run ensemble. Output to single or a common file(s) depends on specification of the value for the out_size_threshold sub-keyword in <model>.cfg. <run_char> is a six-digit placeholder for the corresponding single run number.
During model output post-processing output and structure of results is written to <model>.res<res_char>.[nc | ieee | ascii] and <model>.res<res_char>.[ieee | ascii]. <res_char> is a two-digit placeholder for the number of the result file. It ranges from 01 to 99.
For IEEE and ASCII model output and post-processor output formats, multi-dimensional data is organized in the Fortran column-wise storage model.

12.1 NetCDF Model and Post-Processor Output

The intention for supplying NetCDF format for model and post-processor output is to provide the possibility to generate self-describing, platform-independent data files with metadata that can be interpreted by subsequent visualization techniques. The conventions applied for SimEnv represent a compromise between existing standards and the metadata requirements for a flexible and expressive visualization that is adapted to the requirements of the specific data sets of concern. SimEnv follows the NetCDF Climate and Forecast (NetCDF CF) metadata convention 1.0-beta4. Currently, SimEnv supports only up to 4-dimensional NetCDF output during experiment and post-processor performance.

Model output data types as declared in the model output description file <model>.mdf are transferred into NetCDF data types automatically (check the Table below). By default, post-processor output data is of type float.

SimEnv data type			NetCDF data type
byte	or	int*1	NF_BYTE
short	or	int*2	NF_SHORT
int	or	int*4	NF_INT
float	or	real*4	NF_FLOAT
double	or	real*8	NF_DOUBLE

Tab. 12.1 *NetCDF data types*

12.1.1 Global Attributes

The global attributes used in SimEnv from the CF standard are :institution and :convention. In addition, the following global attributes are defined for model and post-processor output:

Name	Value	Data type
:creation_time	<YYYY-MM-DD HH:MM:SS>	char
:model_name	<model>	char
:model_description	model description according to <model>.mdf	char
:model_description_file	{<path>/}<model>.mdf	char
:experiment_type	[behaviour | Monte-Carlo | sensitivity]	char
:experiment_description	experiment description according to <model>.edf	char
:experiment_description_file	{<path>/}<model>.edf	char
:number_of_runs	<number of runs>	int

Tab. 12.2 *Additional global NetCDF attributes*

12.1.2 Variable Labelling and Variable Attributes

For coordinate variables, two cases of labelling are distinguished:

- If for a given predefined variable, target, model variable or post-processor result one of its coordinates spans the entire range of its general dimension, the already existing coordinate definition is used.
- Otherwise, this concerned coordinate is re-defined using the notation <variable_name>_dim_<coordinate_name>.

The following variable attributes are used according to the CF 1.0-beta4 standard:

Name	Value	Data type
<variable_name>:standard_name	[<coordinate_name> | <predef_coordinate_name> | <predef_var_name> | <target_name> | <variable_name> | <result_name>]	char
<variable_name>:long_name	[<coordinate_description> | <predef_coordinate_description> | <predef_variable_description> | <target_description> | <variable_description> | <result_applied_operator_sequence>]	char
<variable_name>:missing_value	<variable type-depending missing value>	type-dep.
<variable_name>:axis (single coordinate variables only)	[X | Y | Z | T | bin | run | …]	char
<variable_name>:unit	[<coordinate_unit> | <predef_coordinate_unit> | <predef_variable_unit> | <target_unit> | <variable_unit> | <result_unit>]	char
<variable_name>:coordinates (multi-dimensional coordinate variables only)	<par1_lon> <par1_lat>	char
<variable_name>:fill_value	<variable type-depending fill value>	type-dep.

Tab. 12.3 *Variable NetCDF attributes*

- For post-processor output, the **:standard_name attribute** simply counts the number of applied operations because the result name of an arbitrary operation is not known in general. For that reason, the :long_name attribute would re-sample the :standard_name attribute and it is used instead to provide the complete description of the applied operator sequence without defining an additional attribute.
  If macros are included, these are resolved and elementary operations are included only.
- For the **:axis attribute** of a coordinate variable exist defaults.
  For each post-processor result, the first coordinate is assumed to be the „X-axis“, the second and third coordinate are assumed to represent the „Y-“ and „Z-axis“, and the fourth dimension is time T.
  For model results, these attribute values are assigned to coordinate variables describing geographical longitude, geographical latitude, level or height and time. In case other coordinate names are used, these are simply also used for the axis attribute.
- The **:unit attribute** is actually estimated for model output only depending on the description of the corresponding variable keywords in the <model>.mdf file. For post-processing output, it is only used as a placeholder and not calculated from the applied operator sequence so far.
- The **:coordinates attribute** serves to define coordinates depending on other ones and so to allow coordinate transformations. Actually, this attribute is not used.
- Actually, the **:fill_value attribute** is not applied to coordinate variables. It is identically to the :missing_value attribute but open for other definitions.

For visualization requirements, the following additional variable attributes have been defined for SimEnv:

Name	Value	Data type
<variable_name>:monotony (coordinate variables only)	[increasing | decreasing | none]	char
<variable_name>:coo_type	[1 | 2]	integer
<variable_name>:data_range	<min> <max>	char
<variable_name>:index_range_<coordinate> (coordinate variables only)	<min_index> <max_index>	int
<variable_name>:simenv_data_kind	[Predefined Model Variable | Model Target | Model Output Variable | postproc_result]	char
<variable_name>:var_representation	[positions | connections] or both	char
<variable_name>:grid_shift	<shift_x> <shift_y>	real, dimension(2)
<variable_name>:north_pole	<lon_pole> <lat_pole>	real, dimension(2)

Tab. 12.4 *Variable NetCDF attributes for visualization*

- The **:monotony attribute** is applied to coordinate variables only and estimated from the coordinate values as defined in the <model>.mdf file. During post-processing additional coordinates can be generated for which no monotony may be estimated. In such cases, the attribute is set to “none”.
- The **:coo_type attribute** describes the grid representation of a given coordinate. A value of 1 indicates that all coordinate values are provided explicitly (suitable, e.g., for irregular grids). A value of 2 indicates a regular grid and a coordinate representation by its start value, increment and end value.
- The **:data_range attribute** provides the real range that is covered by the related variable in the recent NetCDF file.
- The **:index_range attribute** is used only in case a predefined variable, target, model variable or post-processing result covers not the complete range of a dimension as defined for a coordinate variable. It describes that sub-space for which the concerned target, variable or result is defined.
- The **:var_representation attribute** is introduced to specify what operations are allowed on the data.
- The **:grid_shift attribute** is actually still a placeholder for variables that are not defined in the centre of a grid box when quasi-regular grids are used.
- The **:north_pole attribute** can be used if rotated grids are applied.

12.2 IEEE Compliant Binary Model Output

IEEE compliant binary model output is written in records of fixed length to <model>.out<run_char>.ieee and/or <model>.outall.ieee. Record length is determined by the sub-keyword out_ieee_blocksize and in interrelation to the sub-keyword out_size_threshold in <model>.cfg. For these two sub-keywords and potential modification of the value for out_ieee_blocksize check Tab. 10.1. Sequence of data for each single run is as follows:

- Experiment targets as specified in <model>.edf
  Sequence as in <model>.edf
- Built-in (pre-defined) model output variables
  Sequence as in Tab. 10.8
- Model output variables
  Sequence as in <model>.mdf

Storage demand for each model variable / target is according to its dimensionality, extents and data type. Storage demand in bytes for each model variable / target is readjusted to the smallest number of bytes divisible by 8, where the data can be stored. Multi-dimensional data fields are organized in the Fortran column-wise storage model.
In <model>.outall.ieee each single run starts with a new record. Sequence of single runs corresponds with sequence of the single run numbers <run>: Data from default single run 0 is stored in the first and potentially the following records.

Having a model output description file as in Example 5.1 and an experiment description file as in Example 6.1(a) each single run is stored in the following way:

Target / model variable	Extents	Data type	Storage demand [Byte]	Storage demand adjusted [Byte]
p1	1	float	4	8
p2	1	float	4	8
sim_time	1	float	4	8
atmo	45 x 90 x 4 x 20	float	1.296.000	1.296.000
bios	36 x 90 x 20	float	259.200	259.200
atmo_g	20	int	80	80
bios_g	1	int	4	8
				1.555.312

With out_ieee_blocksize = 100, which transforms to 100*1024 =102.400 Bytes, one single run needs 16 records with a fixed length of 102.400 Bytes. Remaining bytes in the last record are undefined.

Example 12.1 *IEEE compliant model output data structure*

12.3 IEEE Compliant Binary and ASCII Post-Processor Output

For IEEE and ASCII post-processor output result information is stored in two files:

- <model>.res<res_char>.[ieee | ascii] holds the result dynamics
- <model>.inf<res_char>.[ieee | ascii] holds structure and coordinate information

The IEEE post-processor output files <model>.res<res_char>.ieee and <model>.inf<res_char>.ieee are unformatted binary files with IEEE real*4 / integer*4 number representation, while for the ASCII post-processor version <model>.res<res_char>.ascii and <model>.inf<res_char>.ascii formatted ASCII files are used. Files for both output file formats have for each result subsequently the following structure:

Record structure of <model>.inf<res_char>.[ieee | ascii] for each result:

record no. 1	integer*4	length of the string of the operator sequence
	max. char*512	character string of the operator sequence
record no. 2	10 integer*4	idim ext(1) ... ext(9) (ext(i) = 0 for i > idim)
record no. 3	max. char*20	coordinate name of dimension 1
record no. 4 ...	max. 10 real*4	coordinate values of dimension 1 in records of 10 values
...		
record no. xxx	max. char*20	coordinate name of dimension idim
record no. xxx+1 ...	max. 10 real*4	coordinate values of dimension idim in records of 10 values

Record structure of <model>.res<res_char>.[ieee | ascii] for each result:

Record no. 1 ...	max. 10 real*4	result_values(1) ... result_value(length_result)	
		with length_result = product of ext(i) > 0	for idim > 0
		= 1	else

The vector result_value is stored in the Fortran column-wise storage model. The nodata element for undefined result values is set to 3.4E38.

The Fortran code in Example 15.8 reads post-processing binary output files <model>.res<res_char>.ascii and model>.inf<res_char>.ascii in their general structure:

13 Prospects

SimEnv development and improvement is user-driven. Here you can find a list of the main development pathways in future.

General
- Graphical user interface
- Linux and Windows portability
- Unique number representations for binary output (big endians vs. small endians)

Model interface

Experiment preparation
- Experiment types stochastic analysis and gradient-free optimization technique

Experiment performance
- Experiment performance for distributed models across networks

Experiment post-processing
- Experiment specific operators for local sensitivity analysis
- Multi-file model output storage
- Additional advanced operators (netcdf_data, regrid, coarse, sort, ...)
- Advanced uncertainty and global sensitivity analyses operators
- C-interface for user-defined operators
- Flexible assignment of data types to operator results (currently: only real*4)
- Shared memory access (C-shm*-functions) for user-defined operators to avoid data exchange by external files
- Wrapping of pure C/C++-operators in Fortran to use them as built-in operators

Experiment evaluation
- Advanced techniques for graphical representation of post-processor output, especially for multi-run operators

14 References

Bohr, J. (1998) A summary on Probabilities
http://ic.net/~jnbohr/java/CdfDemoMain.html

European Commission – IPSC (2004): SimLab 2.2 Reference Manual
http://www.jrc.ce.eu.int/uasa/primer-sa.asp

Flechsig, M. (1998) SPRINT-S: A Parallelization Tool for Experiments with Simulation Models. PIK-Report No. 47, Potsdam Institute for Climate Impact Research, Potsdam and
http://www.pik-potsdam.de/reports/pr-47/pr47.pdf

Helton, J.C., Davis, F.J. (2000): Sampling-Based Methods.
In: Saltelli *et.al* (2000)

Imam, R.L., Helton, J.C. (1998): An Investigation of Uncertainty and Sensitivity Analysis Techniques for Computer Models. Risk Anal. 8(1), 71-90

Saltelli, A., Chan, K., Scott, E.M. (eds.) (2000) Sensitivity Analysis. J. Wiley & Sons, Chichester

Saltelli, A., Tarantola, S., Campolongo, F., Ratto, M. (2004) Sensitivity Analysis in Practice: A Guide to Assessing Scientific Models. J. Wiley & Sons, Chichester (to appear)

Waszkewitz, J., Lenzen, P., Gillet, N. (2001) The PINGO package: Procedural interface for Grib formatted objects. Max-Planck-Institute for Meteorology, Hamburg and
http://www.mad.zmaw.de/Pingo/pingohome.html

Wenzel, V., Kücken, M., Flechsig, M. (1995) MOSES - Modellierung und Simulation ökologischer Systeme. PIK-Report No. 13, Potsdam Institute for Climate Impact Research, Potsdam

Wenzel, V., Matthäus, E., Flechsig, M. (1990) One Decade of SONCHES. Syst. Anal. Mod. and Sim. 7, 411-428

Wierzbicki, A.P. (1984) Models and Sensitivity of Control Systems. Studies in Automation and Control. Vol. 5. Elsevier, Amsterdam

15 Appendices

15.1 Version Implementation

15.1.1 Linking User Models and User-Defined Operators

For
- User models implemented in C/C++ or Fortran and
- User-defined operators to be used in result post-processing

the following libraries have to be linked to to couple it to the simulation environment
- $SE_HOME/libsimenv.a
- /usr/local/lib/libnetcdf.a

For running models again outside SimEnv check chapter 5.8.

15.1.2 Example Models and User Files

For the following models corresponding files of Tab. 10.4 of can be copied from the corresponding examples-directory of $SE_HOME to the user's working directory by running the SimEnv command simenv.cpy <model> from the working directory:

<model>	Source code	Explanation
world_f	Fortran world_f.f	global atmosphere - biosphere test model at a (lat x lon x level x time) = (45 x 90 x 4 x 20) resolution
world_c	C world_c.c	global atmosphere - biosphere test model at a (lat x lon x level x time) = (45 x 90 x 4 x 20) resolution
world_cpp	C++ world_cpp.cpp	global atmosphere - biosphere test model at a (lat x lon x level x time) = (45 x 90 x 4 x 20) resolution
world_py	Python world_py.py	global atmosphere - biosphere test model at a (lat x lon x level x time) = (45 x 90 x 4 x 20) resolution
world_sh	Shell script level world_sh.f world_shput.f	global atmosphere - biosphere test model at a (lat x lon x level x time) = (45 x 90 x 4 x 20) resolution
world_f_1x1	Fortran world_f_1x1.f	global atmosphere - biosphere test model at a (lat x lon x level x time) = (180 x 360 x 16 x 20) resolution
pixel_f	Fortran Pixel_f.f	global atmosphere - biosphere test model for one lat-lon constellation at a (level x time) = (4 x 20) resolution
gams_model	GAMS gams_model.gms	GAMS model from Example 15.6

Tab. 15.1 *Implemented models for current version*
for <model> = world_* check also Example 1.1

Additionally, the following files are available in the corresponding examples directory of $SE_HOME:

File	Explanation
<model>.[f | c | cpp | py | gms]	model source code
<model>	model executable compiled and linked from <model>.[f | c | cpp]
world.edf_[a | b | c | d | e | f]	experiment description files corresponding to Example 6.1, Example 6.2, and Example 6.3 to be copied to world_[f | c | cpp | py | sh].edf and/or world_f_1x1
world.post_[c | e | bas | adv]	post-processor input file (complete experiment) for world.edf_[c | e] (simenv.res world_[f | c | cpp | py | sh] [new | append | replace] < world.edf_[c | e]) and/or all experiments (selected single run <run>) (simenv.res world_[f | c | cpp | py | sh] [new | append | replace] <run> < world.edf_[bas | adv])
world.dat_[d | e | tab]	data files for world.edf_[d | e] and/or world.post_adv
usr_opr_<opr>.f	source code for user-defined operator <opr>
<opr>.opr	executable for user-defined operator <opr>
model_[f | c | cpp].lnk <model>	compile <model>.[f | c | cpp] and link to executable <model>
usr_opr_<opr>.f	source code file for user-defined post-processing operator <opr>
operator_f.lnk <opr>	compile usr_opr_<opr>.f and link executable <opr>.opr for user-defined post-processing operator <opr>

Tab. 15.2 *Implemented model-related user files for current version*
for <opr> see Tab. 15.3 below

15.1.3 User-Defined Operators

The following user-defined operators are available from the corresponding examples directory of $SE_HOME:

Name	Explanation / restriction	Example
char_test('char1','char2',arg)	character test	see source code
corr_coeff(arg1,arg2)	correlation coefficient R	`corr_coeff(bios,-bios) = -1.`
div(arg1,arg2)	division as an example how the corresponding built in basic operator works	`div(-2,-4) = 0.5`
mat_mul	matrix multiplication of 2-dimensional operands	`mat_mul(mat1,mat2)`
simple_div(arg1,arg2)	division without consideration of overflow, underflow, division by zero	`simple_div(-2,-4) = 0.5`

Tab. 15.3 *Available user-defined operators*

15.1.4 Technical Limitations

Entity		Limitation
User-defined files entities (check also chapter 11)		
max. length of a record in a user-defined file	[characters]	160
max. length of all global descriptions descr	[characters]	512
max. length of a local description descr	[characters]	128
max. length of a unit	[characters]	32
max. length of a directory	[characters]	70
max. length of a record of a referred data file	[characters]	1000
Model interface and experiment preparation entities		
max. length of a name	[characters]	20
max. dimensionality of a model output variable		9
max. dimensionality of a model output variable for Python models		4
max. dimensionality of a model output variable for GAMS models		4
max. dimensionality of a model output variable stored in NetCDF format		4
max. number of model output variables in <model>.mdf		50
max. number of coordinates in <model>.mdf		30
max. number of experiment targets in <model>.edf		50
max. number of slice definitions during interfacing a model		30
max. number of single model runs in an experiment		999.999
max. number of coordinate values and target adjustment values		200.000
max. number of user-defined operators in <model>.odf		45
Post-processing entities (per expression)		
max. number of arguments of an operator		9
max. dimensionality of a result output variable stored in NetCDF format		4
max. number of post-processor output files		99
max. number of characters of an expression		512
max. number of all operands and operators of an expression		200
max. length of a constant	[characters]	20
max. number of constants		30
max. number of allocatable memory segments		10
max. allocatable memory	[MBytes]	240

Tab. 15.4 *Current SimEnv limitations*

15.2 Examples for Model Interfaces, User-Defined Operators, and Result Import Interfaces

15.2.1 Example Implementations for Model world

According to Example 1.1 at page 4 dynamics of the model world depend on four model paramters p1, p2, p3, and p4:

Model target	Target default value	Internal model parameter	Unit	Meaning
p1	1.	phi_lat	π/12	latitudinal phase shift
p2	2.	omega_lat	2*π	latitudinal frequency
p3	3.	phi_lon	π/12	longitudinal phase shift
p4	4.	omega_lon	2*π	longitudinal frequency

Tab. 15.5 *Parameters for the model world*
*Mapping between model targets and internal model parameters is performed by simenv_get_**

For reasons of simplification these parameters influence state variables atmo and bios by the product of two sine terms in the following manner:

```
value_lat(lat) = sin(2*π*omega_lat*f(lat)+phi_lat*π/12)
value_lon(lon) = sin(2*π*omega_lon*f(lon)+phi_lon*π/12)
```

The function f(.) maps lat and/or lon in a way, that yields

```
value_[lat|lon](1)      = sin(+π*omega_[lat|lon]+phi_[lat|lon]*π/12)
value_[lat|lon](last/2) = sin(±0*omega_[lat|lon]+phi_[lat|lon]*π/12)
value_[lat|lon](last)   = sin(-π*omega_[lat|lon]+phi_[lat|lon]*π/12)

atmo(lat,lon,level,time) = value_lat(lat)*value_lon(lon)*(level+100*time)
bios(lat,lon,time)       = value_lat(lat)*value_lon(lon)*100*time

atmo_g(time) = avg(abs(atmo(lat,lon,1,time)))
bios_g       = avg(abs(bios(lat,lon,time)))
```

Average avg is calculated in a box around (lat,lon) = (0°,0°).

15.2.2 Fortran Model

With respect to Example 5.1 the following Fortran code **world_f.f** could be used to describe the model coupled to SimEnv. SimEnv modifications are marked in **bold**.

```
      program world_f
      c  declare SimEnv coupling functions
         integer*4   simenv_ini_f,   simenv_get_f, simenv_get_run_f
         integer*4   simenv_slice_f, simenv_put_f, simenv_end_f
      c  declare atmo without dimensions level and time and bios without time
      c  because they are computed in place and simenv_slice is used
         real*4      atmo(0:44,0:89)
         real*4      bios(0:35,0:89)
         integer*4   atmo_g(0:19)
         integer*4   bios_g
         integer*4   run_int
         character*6 run_char

         istatus = simenv_ini_f()
      c  check return code for the model coupling functions at least here
         if(istatus.ne.0) call exit_(1)
      c  only if necessary:
         istatus = simenv_get_run(run_int,run_char)
         p1 = 1.
         p2 = 2.
         p3 = 3.
         p4 = 4.
         istatus = simenv_get_f('p1',p1,p1)
         istatus = simenv_get_f('p2',p2,p2)
         istatus = simenv_get_f('p3',p3,p3)
         istatus = simenv_get_f('p4',p4,p4)
      c  compute dynamics of atmo and bios over space and time,
      c  of atmo_g over time, all dependent on p1,p2,p3,p4
         do idecade = 0,19
         ...
            do level= 0,3
               istatus = simenv_slice_f('atmo',3,level,level)
               istatus = simenv_slice_f('atmo',4,idecade,idecade)
               istatus = simenv_put_f('atmo',atmo)
            enddo
            istatus = simenv_slice_f('bios',3,idecade,idecade)
            istatus = simenv_put_f('bios',bios)
         enddo
         ...
         istatus = simenv_put_f('atmo_g',atmo_g)
      c  compute dynamics of bios_g
         ...
         istatus = simenv_put_f('bios_g',bios_g)
         istatus = simenv_end_f()
         end
```

Example file:world_f.f

Example 15.1 *Model interface for Fortran models - model world_f.f*

15.2.3 C Model

With respect to Example 5.1 the following C code **world_c.c** could be used to describe the model coupled to SimEnv. SimEnv modifications are marked in **bold**.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/* declare SimEnv coupling functions (compile with -I$SE_HOME) */
#include <simenv.h>

/* declare atmo without dimensions level and time and bios without time*/
/* because they are computed in place and simenv_slice is used */
static float   atmo[45][90];
static float   bios[36][90];
static int     atmo_g[20];
static int     bios_g;

main(void)
{
   float p1,p2,p3,p4;
   int run_int;
   char run_char[6];
   int level,idecade,istatus,idim;
   istatus = simenv_ini_c();
/* check return code of model coupling functions at least here */
   if(istatus != 0) return 1;
/* only if necessary: */
   istatus = simenv_get_run_c(&run_int,run_char);
   p1 = 1.;
   p2 = 2.;
   p3 = 3.;
   p4 = 4.;
   istatus = simenv_get_c('p1',&p1,&p1);
   istatus = simenv_get_c('p2',&p2,&p2);
   istatus = simenv_get_c('p3',&p3,&p3);
   istatus = simenv_get_c('p4',&p4,&p4);
/* compute dynamics of atmo and bios over space and time, */
/* of atmo_g over time, all dependent on p1,p2,p3,p4 */
   for (idecade=0; idecade<=19; idecade++)
   {...
     for (level=0; level<=3; level++)
     { ...
       idim=3;
       istatus = simenv_slice_c('atmo',&idim,&level,&level);
       idim=4;
       istatus = simenv_slice_c('atmo',&idim,&idecade,&idecade);
       istatus = simenv_put_c('atmo',(char *) &atmo);
     }
     idim=3;
     istatus = simenv_slice_c('bios',&idim,&idecade,&idecade);
     istatus = simenv_put_c('bios',(char *) &bios);
   }
   istatus = simenv_put_c('atmo_g',(char *) &atmo_g);
/* compute dynamics of bios_g */
   ...
```

```
        istatus = simenv_put_c('bios_g', ,(char *) &bios_g);
        istatus = simenv_end_c();
        return 0;
    }
```

Example file: world_c.c

Example 15.2 *Model interface for C models – model world_c.c*

15.2.4 C++ Model

With respect to Example 5.1 the following C++ code **world_cpp.cpp** could be used to describe the model coupled to SimEnv. SimEnv modifications are marked in **bold**.

```
#include <stdio.h>
#include <stdlib.h>
/* declare SimEnv coupling functions (compile with -I$SE_HOME) */
#include <simenv.h>

class World
{
/* declare atmo without dimensions level and time and bios without time*/
/* because they are computed in place and simenv_slice is used */
   public:  float   atmo[45][90];
   public:  float   bios[36][90];
   public:  int     atmo_g[20];
   public:  int     bios_g;
   private: int     level,idecade,istatus,idim;

   public: void computeAtmo(float p1 ,float p2, float p3, float p4)
/* compute dynamics of atmo over space and time, */
/* and of atmo_g over time, all dependent on p1,p2,p3,p4 */
   {
     for (idecade=0; idecade<=19; idecade++)
     {...
       for (level=0; level<=3; level++)
       {...
         idim=3;
         istatus = simenv_slice_c('atmo',&idim,&level,&level);
         idim=4;
         istatus = simenv_slice_c('atmo',&idim,&idecade,&idecade);
         istatus = simenv_put_c('atmo',(char *) &atmo);
       }
     }
   }

public: void computeBios(float p1, float p2, float p3, float p4)
/* compute dynamics of bios over space and time, */
/* and of bios_g all dependent on p1,p2,p3,p4 */
   {
     for (idecade=0; idecade<=19; idecade++)
     {...
       idim=3;
       istatus = simenv_slice_c('bios',&idim,&idecade,&idecade);
       istatus = simenv_put_c('bios',(char *) &bios);
     }
/* compute dynamics of bios_g */
     ...
   }

}
```

```
main(void)
{
   int run_int, istatus;
   char run_char[6];
   istatus = simenv_ini_c();
/* check return code of model coupling functions at least here */
   if(istatus != 0) return 1;
/* only if necessary: */
   istatus = simenv_get_run_c(&run_int,run_char);

   float p1 = 1., float p2 = 2., float p3 = 3., float p4 = 4.;
   istatus = simenv_get_c('p1',&p1,&p1);
   istatus = simenv_get_c('p2',&p2,&p2);
   istatus = simenv_get_c('p3',&p3,&p3);
   istatus = simenv_get_c('p4',&p4,&p4);

   World world;
   world.computeAtmo(p1,p2,p3,p4);
   istatus = simenv_put_c('atmo_g',(char *) &(world.atmo_g));
   world.computeBios(p1,p2,p3,p4);
   istatus = simenv_put_c('bios_g',(char *) &(world.bios_g));

   istatus = simenv_end_c();
   return 0;
}
```

Example file: world_cpp.cpp

Example 15.3 *Model interface for C++ models – model world_cpp.cpp*

15.2.5 Python Model

With respect to Example 5.1 the following Python code **world_py.py** could be used to describe the model coupled to SimEnv. SimEnv modifications are marked in **bold**.

```
#!/usr/local/bin/python
import string
import os
from simenv import *
from math import *
from Numeric import *

atmo=zeros([45,90,4,20], Float)
bios=zeros([36,90,20], Float)
atmo_g=zeros([20], Float)
simenv_ini_py()
# only if necessary:
run_int = int(simenv_get_run_py())
p1=1.
p2=2.
p3=3.
p4=4.
p1 = float(simenv_get_py('p1',p1))
p2 = float(simenv_get_py('p2',p2))
p3 = float(simenv_get_py('p3',p3))
p4 = float(simenv_get_py('p4',p4))
# compute dynamics of atmo and bios over space and time,
# of atmo_g over time, all dependent on p1,p2,p3,p4
for idecade in range(20):
...
   for level in range(4):
   ...
atmo=reshape(atmo,45*90*4*20,))
simenv_put_py('atmo',atmo)
bios=reshape(atmo,45*90*20,))
simenv_put_py('bios',bios)
simenv_put_py('atmo_g',atmo_g)
# compute dynamics of bios g
# ...
simenv_put_py('bios_g',bios_g)
simenv_end_py()
```

Example file:world_py.py

Example 15.4 *Model interface for Python models – model world_py.py*

15.2.6 Model Interface at Shell Script Level

Assume any experiment. Assume model executable world_sh to target values p1 to p4 as arguments from the command line.
The shell script **world_sh.run** with an interface at shell script level to run the model world_sh and to transform model output to SimEnv could look like:

```
# always perform at begin
. $SE_HOME/simenv_ini_sh

# create temporary directory run<run char> to perform the model
# and model output transformation from native to SimEnv structure there
. $SE_HOME/simenv_get_run_sh
mkdir run$run_char
cd run$run char

# get adjustments for p1 ... p4
target_name='p1'
target_def_val=1.
. $SE_HOME/simenv_get_sh
target_name='p2'
target_def_val=2.
. $SE_HOME/simenv_get_sh
target_name='p3'
target_def_val=3.
. $SE_HOME/simenv_get_sh
target_name='p4'
target_def_val=4.
. $SE_HOME/simenv_get_sh

# run the model
cp ../land_sea_mask.coarsed .
../world_sh $p1 $p2 $p3 $p4

# read model results and output them to SimEnv
../world_shput

# clear and remove directory
rm -f *
cd ..
rmdir run$run_char

# always perform at end
. $SE_HOME/simenv_end_sh
```

Example file: world_sh.run

Example 15.5 *Model interface at shell script level – model shell script world_sh.run*

15.2.7 GAMS Model

The SimEnv version comes with a coupled GAMS model **gams_model.gms** and all associated files that fully correspond with the GAMS example model at http://www.gams.com/docs/gams/Tutorial.pdf. Modifications for SimEnv are marked in **bold**.

```
SETS
   I     canning plants   / SEATTLE, SAN-DIEGO /
   J     markets          / NEW-YORK, CHICAGO, TOPEKA / ;
PARAMETERS
   A(I)  capacity of plant i in cases
     /    SEATTLE     350
          SAN-DIEGO   600  /
   B(J)  demand at market j in cases
     /    NEW-YORK    325
          CHICAGO     300
          TOPEKA      275  / ;

* - Before using parameter (here: dem_ny and dem_ch) as SimEnv experiment
*   targets they have to be declared as GAMS parameters with their
*   default values from above.
* - then insert $include <model> simenv get.inc
*   simenv_get.inc is generated automatically based on <model>.edf
* - and assign adjusted targets to model variables
   PARAMETERS
   dem_ny /325.0/;
   dem_ch /300.0/;
   $include gams_model_simenv_get.inc
   A("SEATTLE")   = dem_ny;
   A("SAN-DIEGO") = dem_ch;

TABLE  D(I,J)  distance in thousands of miles
                NEW-YORK       CHICAGO      TOPEKA
   SEATTLE          2.5           1.7          1.8
   SAN-DIEGO        2.5           1.8          1.4 ;
SCALAR F  freight in dollars per case per thousand miles  /90/

* get the model status as a model output
   modstat is set to transport.modelstat ;

PARAMETER C(I,J)  transport cost in thousands of dollars per case ;
   C(I,J) = F * D(I,J) / 1000 ;
VARIABLES
   X(I,J)  shipment quantities in cases
   Z       total transportation costs in thousands of dollars ;
POSITIVE VARIABLE X ;
EQUATIONS
   COST        define objective function
   SUPPLY(I)   observe supply limit at plant i
   DEMAND(J)   satisfy demand at market j ;
COST ..        Z  =E=  SUM((I,J), C(I,J)*X(I,J)) ;
SUPPLY(I) ..   SUM(J, X(I,J))  =L=  A(I) ;
DEMAND(J) ..   SUM(I, X(I,J))  =G=  B(J) ;
MODEL TRANSPORT /ALL/ ;
SOLVE TRANSPORT USING LP MINIMIZING Z ;
```

```
* After solving the equations $include simenv_put.inc
* has to be inserted.
* simenv_put.inc is generated automatically by SimEnv
* based on <model>.edf and <model>.gdf
* Additional GAMS commands are possible after the last modification
   modstat = transport.modelstat
   $include gams_model_simenv_put.inc
```

Example file:gams_model.gms

Example 15.6 *Model interface for GAMS models – model gams_model.gms*

15.2.8 User-Defined Operator

Implementation of the user-defined operator mat_mul in the file **usr_opr_mat_mul.f**:

```
      function icheck_user_def_operator()
      c  declare fields to hold extents and coordinates
         dimension iext1(9),iext2(9)
         dimension ico_blk1(9),ico_blk2(9)
         dimension ico_beg1(9),ico_beg2(9)

      c  get dimensionality idimens, extents iext,
      c  coordinate block number ico_blk and
      c  begin number for coordinates ico_beg in coordinate block ico_blk
         idimens1=iget_dim_arg(1,iext1)
         idimens2=iget dim arg(2,iext2)
         iok=iget_co_arg(1,ico_blk1,ico_beg1)
         iok=iget_co_arg(2,ico_blk2,ico_beg2)
      c  get check modus for coordinates
         ichk modus=iget co chk modus()

         if(idimens1.ne.2.or.idimens2.ne.2) then
      c  wrong dimensionalities
            ierror=1
         else
            if(iext1(2).ne.iext2(1)) then
      c  wrong extents
               ierror=2
            else
               if(ico_blk1(2).eq.ico_blk2(1)) then
      c  coordinates identical
                  if(ico_beg1(2).eq.ico_beg2(1)) then
                     iret=31
                  else
                     iret=33
                  endif
               else
      c  differing coordinates
                  iret=32
                  if(ichk_modus.eq.1) then
      c  check only for weak coordinate
                     do j=0,iext1(2)-1
      c  get coordinate values
                        iretv1=iget_coord_val(
           #                   ico_blk1(2),ico_beg1(2)+j,value1)
                        iretv2=iget_coord_val(
           #                   ico_blk2(1),ico_beg2(1)+j,value2)
      c  iret=33: differing coordinate values
                        if(value1.ne.value2) iret=33
                     enddo
                  endif
               endif
               ierror=0
               if(ichk_modus.eq.2) then
                  if(iret.gt.31) ierror=3
               elseif(ichk_modus.eq.1) then
                  if(iret.gt.32) ierror=3
               endif
            endif
         endif
```

```
      if(ierror.eq.0) then
        iext1(2)=iext2(2)
        ico_blk1(2)=ico_blk2(2)
        ico_beg1(2)=ico_beg2(2)
        iok=iput_struct_res(0,idimens1,iext1,ico_blk1,ico_beg1)
      endif

c     return error code
      icheck_user_def_operator=ierror
      return
      end

      function icompute_user_def_operator(res)
c     SimEnv operator results are always of type real*4
      real*4 res(1)
c     auxiliary variables
      integer*4 iext1(9),iext2(9)
      real*8 value8

c     get dimensionality idimens and extents iext for both arguments
      idimens=iget_dim_arg(1,iext1)
      idimens=iget_dim_arg(2,iext2)

c     perform matrix multiplication
      m=0
      do k=1,iext2(2)
         ioffs2=(k-1)*iext2(1)
         do i=1,iext1(1)
            ioffs1=i
c      res(i,k) = sum(arg1(i,l) * arg2(l,k))
            value8=0.
            indi_defined=0
            do l=1,iext1(2)
               ia1=ioffs1+(l-1)*iext1(1)
               ia2=ioffs2+l
               fac1=arg1(ia1)
               fac2=arg2(ia2)
               if(is_undef(fac1)+is_undef(fac2).eq.0) then
                  indi_defined=1
                  value8=value8+fac1*fac2
               endif
            enddo
            m=m+1
            if(indi defined.eq.0) then
               res(m)=set_undef()
            else
               res(m)=clip_undef(value8)
            endif
         enddo
      enddo

c     return error code
      icompute_user_def_operator=0
      return
      end
```

Example file: usr_opr_mat_mul.f

Example 15.7 *User-defined operator module – operator mat_mul*

15.2.9 Result Import Interface

Implementation of an interface to import post-processor output from SimEnv in ASCII format:

```
         program read_ieee_file
         real*4, pointer, dimension(:) :: coord_values
         real*4, pointer, dimension(:) :: result_values
         integer*4 name_length,idummy,idim, iext(9), length_values
         character result_name*512, coord_name*20
         open(unit=1,file='world_f.inf03.ascii',form='formatted',status='old')
         open(unit=2,file='world_f.res03.ascii',form='formatted',status='old')
      iostat=0
         do while (iostat.eq.0)
            read(1,'(i8)',iostat=iostat) name_length
            if(iostat.eq.0) then
               backspace(1)
               read(1,11) idummy,result_name(1:name_length)
      11       format(i8,a<name_length>)
               read(1,'(10i8)') idim,(iext(i),i=1,9)
               length_result=1
               do i=1,idim
      c           result no. res<i>:
                  length_result=length_result*iext(i)
                  read(1,'(a20)') coord_name
                  allocate(coord_values(iext(i)))
                  ibeg=1
                  do while (ibeg.le.iext(i))
                     iend=min0(ibeg+9,iext(i))
                     read(1,'(10g12.6)') (coord_values(i),i=ibeg,iend)
                  enddo
      c           further processing of coordinate values
                  deallocate (coord_values)
               enddo
               allocate(result_values(length_result))
               ibeg=1
               do while (ibeg.le.length_result)
                  iend=min0(ibeg+9,length_result)
                  read(2,'(10g12.6)') (result_values(i),i=ibeg,iend)
               enddo
      c        further processing of result values
               deallocate (result_values)
            endif
         enddo
         close(unit=1)
         close(unit=2)
         end
```

Example 15.8 *ASCII compliant post-processor export interface*

15.3 Post-Processor Built-in Operators (in Thematic Order)

arg general numerical argument
const_arg constant value argument
char_arg character argument

Name	Meaning
Elemental operators	
arg1 + arg2	addition
arg1 - arg2	subtraction
arg1 * arg2	multiplication
arg1 / arg2	division
arg1 ** arg2	exponentiation
+ arg	identity
- arg	negation
(arg)	parentheses
Basic operators	
abs(arg)	absolute value
dim(arg1,arg2)	positive difference
exp(arg)	exponentiation
int(arg)	truncation value
log(arg)	natural logarithm
log10(arg)	decade logarithm
mod(arg1,arg2)	remainder
nint(arg)	round value
sign(arg)	sign of value
sqrt(arg)	square root
Trigonometric operators	
sin(arg)	sine
cos(arg)	cosine
tan(arg)	tangent
cot(arg)	cotangent
asin(arg)	arc sine
acos(arg)	arc cosine
atan(arg)	arc tangent
acot(arg)	arc cotangent
sinh(arg)	hyperbolic sine
cosh(arg)	hyperbolic cosine
tanh(arg)	hyperbolic tangent
coth(arg)	hyperbolic cotangent
Miscellaneous operators	
classify(const_arg,arg)	classification of arg into const_arg classes
clip(char_arg,arg)	clip arg according to char_arg
cumul(char_arg,arg)	cumulates arg according to char_arg
experiment(char_arg1, char_arg2,arg)	include an other experiment
flip(char_arg,arg)	flip arg according to char_arg
if(char_arg,arg1,arg2,arg3)	general purpose conditional if-construct

Name	Meaning
mask(char_arg,arg1,arg2)	mask elements of argument arg1
matmul(arg1,arg2)	matrix multiplication
nr_of_runs()	number of single runs of the current experiment
rank(char_arg,arg)	rank of arg according to char_arg
run(char_arg,arg)	values of arg for a single run selected by char_arg
table_fct(char_arg,arg)	table function with linear interpolation of table char_arg for position arg
transpose(char_arg,arg)	transpose arg according to char_arg
undef()	undefined element
Aggregation and moment operators for arguments	
min(arg)	argument minimum of values
max(arg)	argument maximum of values
sum(arg)	argument sum of values
avg(arg)	argument linear mean of values
var(arg)	argument variance of values
avgg(arg)	argument geometric mean of values
avgh(arg)	argument harmonic mean of values
avgw(arg1,arg2)	argument weighted mean of values
hgr(const_arg,arg)	argument histogram of values
count(char_arg,arg)	count number of values
minprop(arg)	index of the element where the minimum is reached the first time
maxprop(arg)	index of the element where the maximum is reached the first time
Multiple aggregation and moment operators for arguments	
min_n(arg1,...,argn)	minimum per element
max_n(arg1,...,argn)	maximum per element
minprop_n(arg1,...,argn)	argument position (1 ... n) where the minimum is reached the first time
maxprop_n(arg1,...,argn)	argument position (1 ... n) where the maximum is reached the first time
Dimension-related aggregation and moment operators for arguments	
min_l(char_arg,arg)	dimension-related argument minima of values
max_l(char_arg,arg)	dimension-related argument maxima of values
sum_l(char_arg,arg)	dimension-related argument sums of values
avg_l(char_arg,arg)	dimension-related argument linear means of values
var_l(char_arg,arg)	dimension-related argument variances of values
avgg_l(char_arg,arg)	dimension-related argument geometric means of values
avgh_l(char_arg,arg)	dimension-related argument harmonic means of values
avgw_l(char_arg,arg1,arg2)	dimension-related argument weighted means of values
hgr_l(char_arg,const_arg,arg)	dimension-related argument histograms of values
count_l(char_arg1,char_arg2, arg)	dimension-related count numbers of values
minprop_l(char_arg,arg)	dimension-related argument position (1 ... n) where the minimum is reached the first time
maxprop_l(char_arg,arg)	dimension-related argument position (1 ... n) where the maximum is reached the first time
Multi-run operators (behavioural analysis)	
behav(char_arg,arg)	general purpose operator for navigating and aggregating in the experiment space

Name	Meaning
Multi-run operators (Monte-Carlo analysis)	
cnf(const_arg,arg)	positive distance of confidence line from average avg_e(arg)
cor(arg1,arg2)	correlation coefficient between arg1 and arg2
cov(arg1,arg2)	covariance between arg1 and arg2
ens(arg)	whole Monte-Carlo run ensemble
krt(arg)	kurtosis (4^{th} moment)
med(arg)	median
qnt(const_arg,arg)	quantile of arg
reg(arg1,arg2)	linear regression coefficient to forecast arg2 from arg1
rng(arg)	range = max_e(arg) - min_e(arg)
skw(arg)	skewness (3^{rd} moment)
stat(const_arg1,const_arg2, const_arg3,const_arg4,arg5)	basic statistical summaries
min_e(arg)	run ensemble minimum
max_e(arg)	run ensemble maximum
sum_e(arg)	run ensemble sum
avg_e(arg)	run ensemble average
var_e(arg)	run ensemble variance
avgg_e(arg)	run ensemble geometric average
avgh_e(arg)	run ensemble harmonic average
avgw_e(arg1,arg2)	run ensemble weighted average
hgr_e(const_arg,arg)	heuristic probability density function
count_e(char_arg,arg)	run ensemble count number of values
minprop_e(arg)	run number where the minimum is reached the first time
maxprop_e(arg)	run number where the maximum is reached the first time

Tab. 15.6 *Post-processor built-in operators (in thematic order)*

15.4 Post-Processor Built-in Operators (in Alphabetic Order)

arg general numerical argument
const_arg constant value argument
char_arg character argument

Name	Meaning
arg1 + arg2	addition
arg1 - arg2	subtraction
arg1 * arg2	multiplication
arg1 / arg2	division
arg1 ** arg2	exponentiation
+ arg	identity
- arg	negation
(arg)	parentheses
abs(arg)	absolute value
acos(arg)	arc cosine
acot(arg)	arc cotangent
asin(arg)	arc sine
atan(arg)	arc tangent
avg(arg)	argument linear mean of values
avg_e(arg)	run ensemble average
avg_l(char_arg,arg)	dimension-related argument linear means of values
avgg(arg)	argument geometric mean of values
avgg_e(arg)	run ensemble geometric average
avgg_l(char_arg,arg)	dimension-related argument geometric means of values
avgh(arg)	argument harmonic mean of values
avgh_e(arg)	run ensemble harmonic average
avgh_l(char_arg,arg)	dimension-related argument harmonic means of values
avgw(arg1,arg2)	argument weighted mean of values
avgw_e(arg1,arg2)	run ensemble weighted average
avgw_l(char_arg,arg1,arg2)	dimension-related argument weighted means of values
behav(char_arg,arg)	general purpose operator for navigating and aggregating in the experiment space
classify(const_arg,arg)	classification of arg into const_arg classes
clip(char_arg,arg)	clip arg according to char_arg
cnf(const_arg,arg)	positive distance of confidence line from average avg_e(arg)
cor(arg1,arg2)	correlation coefficient between arg1 and arg2
cos(arg)	cosine
cosh(arg)	hyperbolic cosine
cot(arg)	cotangent
coth(arg)	hyperbolic cotangent
count(char_arg,arg)	count number of values
count_e(char_arg,arg)	run ensemble count
count_l(char_arg1,char_arg2, arg)	dimension-related count numbers of values
cov(arg1,arg2)	covariance between arg1 and arg2
cumul(char_arg,arg)	cumulates arg according to char_arg
dim(arg1,arg2)	positive difference
ens(arg)	whole Monte-Carlo run ensemble
exp(arg)	exponentiation
experiment(char_arg1,	include an other experiment

Name	Meaning
char_arg2,arg)	
flip(char_arg,arg)	flip arg according to char_arg
hgr(const_arg,arg)	argument histogram of values
hgr_e(const_arg,arg)	heuristic probability density function
hgr_l(char_arg,const_arg,arg)	dimension-related argument histograms of values
if(char_arg,arg1,arg2,arg3)	general purpose conditional if-construct
int(arg)	truncation value
krt(arg)	kurtosis (4^{th} moment)
log(arg)	natural logarithm
log10(arg)	decade logarithm
mask(char_arg,arg1,arg2)	mask elements of argument arg1
matmul(arg1,arg2)	matrix multiplication
max(arg)	argument maximum of values
max_e(arg)	run ensemble maximum
max_l(char_arg,arg)	dimension-related argument maxima of values
max_n(arg1,...,argn)	maximum per element
maxprop(arg)	index of the element where the maximum is reached the first time
maxprop_e(arg)	run number where the maximum is reached the first time
maxprop_l(char_arg,arg)	dimension-related argument position (1 ... n) where the maximum is reached the first time
maxprop_n(arg1,...,argn)	argument position (1 ... n) where the maximum is reached the first time
med(arg)	median
min(arg)	argument minimum of values
min_e(arg)	run ensemble minimum
min_l(char_arg,arg)	dimension-related argument minima of values
min_n(arg1,...,argn)	minimum per element
minprop(arg)	index of the element where the minimum is reached the first time
minprop_e(arg)	run number where the minimum is reached the first time
minprop_l(char_arg,arg)	dimension-related argument position (1 ... n) where the minimum is reached the first time
minprop_n(arg1,...,argn)	argument position (1 ... n) where the minimum is reached the first time
mod(arg1,arg2)	remainder
nint(arg)	round value
nr_of_runs()	number of single runs of the current experiment
qnt(const_arg,arg)	quantile of arg
rank(char_arg,arg)	rank of arg according to char_arg
reg(arg1,arg2)	linear regression coefficient to forecast arg2 from arg1
rng(arg)	range = max_e(arg) - min_e(arg)
run(char_arg,arg)	values of arg for a single run selected by char_arg
sign(arg)	sign of value
sin(arg)	sine
sinh(arg)	hyperbolic sine
skw(arg)	skewness (3^{rd} moment)
sqrt(arg)	square root
stat(const_arg1,const_arg2, const_arg3,const_arg4,arg5)	basic statistical summaries
sum(arg)	argument sum of values
sum_e(arg)	run ensemble sum
sum_l(char_arg,arg)	dimension-related argument sums of values
table_fct(char_arg,arg)	table function with linear interpolation of table char_arg for position arg
tan(arg)	tangent
tanh(arg)	hyperbolic tangent
transpose(char_arg,arg)	transpose arg according to char_arg

Name	Meaning
undef()	undefined element
var(arg)	argument variance of values
var_e(arg)	run ensemble variance
var_l(char_arg,arg)	dimension-related argument variances of values

Tab. 15.7 *Post-processor built-in operators (in alphabetical order)*

15.5 Character Arguments of Built-in Operators

Tab. 15.8 summarises for built-in operators character argument values. User-defined operators can not have pre-defined character argument values.

Operator	Argument number	Argument value (without quotation marks, pre-defined values are case-insensitive)
avg_l	1	sequence of digits 0 and 1
avgg_l	1	sequence of digits 0 and 1
avgh_l	1	sequence of digits 0 and 1
avgw_l	1	sequence of digits 0 and 1
behav	1	(not pre-defined, case insensitive)
clip	1	(not pre-defined, case insensitive)
count	1	[all | def | undef]
count_e	1	[all | def | undef]
count_l	1	sequence of digits 0 and 1
count_l	2	[all | def | undef]
cumul	1	sequence of digits 0 and 1
experiment	1	(not pre-defined, case sensitive)
experiment	2	(not pre-defined, case insensitive)
flip	1	sequence of digits 0 and 1
hgr_l	1	sequence of digits 0 and 1
if	1	[< | <= | > | >= | = | != | def | undef]
mask	1	[< | <= | > | >= | = | !=]
max_l	1	sequence of digits 0 and 1
maxprop_l	1	sequence of digits 0 and 1
min_l	1	sequence of digits 0 and 1
minprop_l	1	sequence of digits 0 and 1
rank	1	[tie_plain | tie_min | tie_avg]
run	1	[run number | not pre-defined]
sum_l	1	sequence of digits 0 and 1
table_fct	1	(not pre-defined, case sensitive)
transpose	1	sequence of digits 1 to 9
var_l	1	sequence of digits 0 and 1

Tab. 15.8 *Character arguments of built-in operators*

The length of the character string argument with a sequence of digits corresponds with the dimensionality of the non-character argument under investigation.

15.6 Constant Arguments of Built-in Operators

Tab. 15.8 summarises for built-in operators constant argument values. User-defined operators can not have pre-defined constant argument values.

Operator	Argument number	Argument type	Argument value
classify	1	integer	[0 | ≥ 2]
cnf	1	real	[0.001 | 0.01 | 0.05 | 0.1]
hgr	1	integer	[0 | ≥ 4]
hgr_e	1	integer	[0 | ≥ 4]
hgr_l	2	integer	[0 | ≥ 4]
qnt	1	real	0. ≤ arg1 ≤ 100.
stat	1	real	0. ≤ arg1 ≤ 100.
stat	2	real	0. ≤ arg2 ≤ 100.
stat	3	real	[0.001 | 0.01 | 0.05 | 0.1]
stat	4	real	[0.001 | 0.01 | 0.05 | 0.1]

Tab. 15.9 *Constant arguments of built-in operators*

15.7 Glossary

The glossary defines terms in that sense they are used in this User's Guide.
An arrow → refers to another term in the glossary.

Adjustment: Numerical modification of a → target during an → experiment. Adjustments are related to an → experiment type and are described in the experiment description → user-defined file.

ASCII: The **A**merican **S**tandard **C**ode for **I**nformation and **I**nterchange developed by the American National Standards Institute (http://www.ansi.org) is used in SimEnv to store information in → user-defined files.

Behavioural analysis: → Experiment type to inspect behaviour of a → model in a space, spanned up by → targets. The target space is scanned in a deterministic manner, applying pre-defined → adjustments of the targets with a flexible scanning strategy for target sub-spaces.

Coordinate coord: In the model description → user-defined file each → dimension of a → variable with a → dimensionality greater than 0 a coordinate is assigned to. A coordinate has a unique name and strictly monotonic ordered coordinate values. The number of coordinate values corresponds with the → extent for this dimension. Consequently, each model variable with a dimensionality greater than 0 resides at a assigned (multi-dimensional) → grid.

Coupling: Coupling means interfacing a → model to SimEnv and enabling finally experimenting with a model within SimEnv. There are coupling interfaces at programming language level for C/C++, Fortran, → Python, and → GAMS. Additionally, models can be coupled at the shell script level by using shell script syntax elements. For all coupling techniques, the model is wrapped into a shell script.

Data type: The type of a → variable as declared in the → model and the corresponding model description → user-defined file. SimEnv data types are byte, short, int, float, and double.

Default value: The nominal (standard) numerical value of an experiment → target. The default value is specified in the experiment description → user-defined file and for → coupling at the language level also in the model code.

Dimension: → dimensionality

Dimensionality dim: The number of dimensions of a model → variable or of an → operand in model output post-processing. In the model description → user-defined file each variable a dimensionality is assigned to that corresponds with the dimensionality of the related model output field in the model source code. Dimensionality 0 corresponds to a scalar, dimensionality 1 to a vector, dimensionality 2 to a matrix.

Environment variable: At UNIX operating system level the so called environment is set up as an array of operating-system and user-defined environment variables that have the form Name=Value. The Value of a Name can be addressed by $Name. In SimEnv directory and path strings or parts of it with environment variables are forbidden.

Experiment: Performing simulation runs with a → model in a co-ordinated manner by applying → experiment types and running the model in a run ensemble, i.e., a series of single simulation runs.

Experiment target: → target

Experiment type: Pre-defined multi-run simulation experiment. In the process of experiment preparation (defining an experiment by describing it in the experiment description → user-defined file) → targets are assigned to an experiment type and experiment-specific → adjustments and other information are assigned to the targets. Currently available experiment types are → behavioural analysis, → Monte-Carlo analysis, and → local sensitivity analysis.

Extent ext: The number of values for a dimension (from the → dimensionality) of a model → variable or of an → operand in model output post-processing. Extents are always greater than 1. Model variables and operands of dimensionality 0 do not have an extent.

Fortran column-wise storage model: A rule how to map a multi-dimensional data field to a 1-dimensional vector. A data field field(1:ext_1 , 1:ext_2 , ... , 1:ext_{dim}) of → dimensionality dim and → extents ext_1, ext_2, ..., ext_{dim} is mapped in Fortran in the following way on a 1-dimensional vector vector(1:ext_1 * ext_2 * ... * ext_{dim})

```
ipointer=0
do i_dim = 1 , ext_dim
  ...
    do i_2 = 1 , ext_2
        do i_1 = 1 , ext_1
            ipointer = ipointer + 1
            vector(ipointer) = field(i_1 , i_2 , ... , i_dim)
        enddo
    enddo
  ...
enddo
```

GAMS: The **G**eneral **A**lgebraic **M**odeling **S**ystem (http://www.gams.com) is a high-level modeling system for mathematical programming problems. It consists of a language compiler and a stable of integrated high-performance solvers. GAMS is tailored for complex, large scale modeling applications, and allows to build large maintainable models that can be adapted quickly to new situations.

Grid: Regular topological structure for a model → variable or an → operator result in post-processing, spanned up as the Cartesian product of the assigned → coordinates to the variable or the operator result.

IEEE: SimEnv can use on demand for storage of model and post-processor output the **I**nstitute of **E**lectrical and **E**lectronics **E**ngineers (http://www.ieee.org) standard #754 for binary storage of floating point numbers.

Macro: An abbreviation for a unique expression, formed from a chain of operands and operators to apply during post-processing. Macros can be embedded into other expressions and are plugged into the expression during its evaluation. Macros are described in the macro description → user-defined file.

Model: A model is a deterministic or stochastic algorithm, coded in one or a number of computer programs that transforms a sequence of input values (→ targets) into a sequence of output values (→ variables). Normally, inputs are parameters, driving forces, initial values, or boundary values to the model, outputs are state variables of the model. For many cases, the model will be state deterministic, time and space dependent. For SimEnv, the model, its targets and variables are interfaced in the process of model → coupling.

Model coupling: → coupling

Model interface: → coupling

Model output post-processing operator: → operator

Model variable: → variable

Monte-Carlo analysis: → Experiment type with pre-single run perturbations of experiment → targets. Each perturbed target a → probability density function pdf with function parameters is assigned to. During the → experiment → adjustments of the targets are realizations from the pdf's using random number techniques. In experiment post-processing statistical measures can be derived from model output of the run ensemble. A prominent statistical measure is the heuristic pdf (histogram) of a model → variable and its relation to the pdf's of the targets.

NetCDF: **Net**work **C**ommon **D**ata **F**orm is an interface for array-oriented data access and a library that provides an implementation of the interface. The NetCDF library also defines a machine-independent format for representing scientific data. Together, the interface, library, and format support the creation, access, and sharing of scientific data. The NetCDF software was developed at the Unidata Program Center in Boulder, Colorado (http://www.unidata.ucar.edu). NetCDF is freely available. SimEnv follows for model output and post-processing output storage the NetCDF Climate and Forecast (CF) metadata convention 1.0-beta4 (http://www.cgd.ucar.edu/cms/eaton/cf-metadata/index.html) and extends it.

OpenDX: The **Open D**ata E**x**plorer OpenDX (http://www.opendx.org) is a uniquely full-featured open source project and software package for the visualization of scientific, engineering and analytical data: Its open system design is built on a standard interface environment. The data model provides users with great flexibility in creating visualizations. OpenDX is based on IBM's Visualization Data Explorer.

Operand: Argument of an → operator in SimEnv model output post-processing. An operand can be a model → variable, an experiment → target, a constant, a character string, → a macro and an operator.

Operator: Computational algorithm how to transform the values of a sequence of → operands into a sequence of operator results during model output post-processing. An operator transforms → dimensionality, → extents, and → coordinates from the operands into the corresponding information for the operator result. There are built-in elemental, basic, and advanced operators as well as built-in operators related to specific → experiment types. Additionally, SimEnv offers specification of user-defined operators according to an operator interface. User-defined operators are announced to the system in the operator description → user-defined file.

Probability density function pdf: A probability density function serves to represent a probability distribution in terms of integrals. A probability distribution assigns to every interval of real numbers a probability.

Python: Python (http://www.python.org) is an portable, interpreted, interactive, object-oriented programming language. It incorporates modules, exceptions, dynamic typing, and very high level dynamic data types, and classes.

Local sensitivity analysis: → Experiment type with incremental → adjustments of → targets in the neighbourhood of the → default values of the targets. A local sensitivity analysis in SimEnv is always performed independently for all targets involved.

Target: Element of the input set of a → model. Targets are manipulated numerically during an → experiment. Targets can be addressed in model output post-processing and they have there a → dimensionality of zero.

Target adjustment: → adjustment

User-defined files: A set of → ASCII files to describe → model-, → experiment-, → operator-, → macro-, and → GAMS model specific information and to determine general SimEnv settings. All user-defined files follow the same syntax rules.

Variable: Element of the output set of a → model that is stored in a SimEnv model or post-processor output format. Variables are defined in the model as well as in the appropriate model description → user file. Each variable has a unique → data type, a → dimensionality, → extents and an assigned → grid. Normally, variable consists of a series of values, forming fields.

White spaces: → ASCII characters space (blank) and horizontal tabulator used in → user-defined files or within expressions in model output post-processing.

Working directory: The directory, a SimEnv service was started from.