The Multi-Run Simulation Environment

SimEnv

User Guide for Version 1.22 (23-Jun-2005)

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

The Multi-Run Simulation Environment

SimEnv

User Guide for Version 1.22 (23-Jun-2005)

by

Michael Flechsig	Potsdam Institute for Climate Impact Research Department Data & Computation, flechsig@pik-potsdam.de
Uwe Böhm	Potsdam Institute for Climate Impact Research Climate System Department, 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 on the Internet:
http://www.pik-potsdam.de/software/simenv/

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 Rostock Institute of Computer Graphics
Albert-Einstein-Str. 21
18059 Rostock, Germany

Phone	++49 – 381 – 498 7481
Fax	++49 – 331 – 498 7482
WWW	http://wwwicg.informatik.uni-rostock.de

Contents

EXECUTIVE SUMMARY 1

1 ABOUT THIS DOCUMENT 3
1.1 Document Conventions 3
1.2 Example Layout 4

2 GETTING STARTED 5

3 VERSION 1.22 7
3.1 What is New? 7
3.2 Limitations / Problems and Their Workarounds 8
3.3 Known Bugs and Their Workarounds 9

4 EXPERIMENT TYPES 11
4.1 General Approach 11
4.2 Behavioural Analysis 13
4.3 Monte Carlo Analysis 14
4.4 Local Sensitivity Analysis 16
4.5 Optimization 18

5 MODEL INTERFACE 21
5.1 General Approach 21
5.2 Grid and Coordinate Assignments to Variables 24
5.3 Model Output Description File <model>.mdf 25
5.4 Model Interface for Fortran and C/C++ Models 27
5.5 Model Interface for Python Models 30
5.5.1 Standard Shell Scripts for Python Models 31
5.6 Model Interface at Shell Script Level 31
5.7 Model Interface for GAMS Models 33
5.7.1 Standard Shell Scripts for GAMS Models 34
5.7.2 GAMS Description File <model>.gdf 36
5.7.3 Files Created during GAMS Model Performance 38
5.8 Semi-Automated Model Interface 39
5.9 Supported Model Structures 40
5.10 Using Interfaced Models Outside SimEnv 41

6 EXPERIMENT PREPARATION 43
6.1 General Approach - Experiment Description File <model>.edf 43
6.2 Behavioural Analysis 44
6.2.1 The Combination 45
6.2.2 Example 46
6.3 Monte Carlo Analysis 47
6.3.1 Distribution Functions and their Parameters 47
6.3.2 Example 48
6.4 Local Sensitivity Analysis 49
6.4.1 Sensitivity Functions and Run Sequence 49
6.4.2 Example 50
6.5 Optimization 51
6.5.1 Special Features in Optimization 51
6.5.2 Example 52

7 EXPERIMENT PERFORMANCE 53
7.1 General Approach 53
7.2 Model Wrap Shell Script <model>.run, Experiment-Specific Preparation and Wrap-Up Shell Scripts 54
7.3 Experiment Parallelization 55
7.4 Experiment Restart 56
7.5 Experiment Partial Performance 58
7.6 Experiment Related User Shell Scripts and Files 59
7.7 Saving Experiments 61

8 EXPERIMENT POST-PROCESSING 63
8.1 General Approach 63
8.1.1 Post-Processor Results 63
8.1.2 Operands 64
8.1.3 Model Output Variables 65
8.1.4 Operators 67

8.1.5 Operator Classification, Flexible Coordinate Checking ... 68
8.2 Built-In Generic Standard Aggregation / Moment Operators ... 70
8.3 Built-In Elemental, Basic, and Advanced Operators ... 70
8.3.1 Elemental Operators ... 70
8.3.2 Basic and Trigonometric Operators ... 71
8.3.3 Standard Aggregation / Moment Operators ... 72
8.3.4 Advanced Operators ... 75
8.3.5 Examples ... 81
8.4 Built-In Experiment Specific Operators ... 82
8.4.1 Standard Aggregation / Moment Operators ... 83
8.4.2 Behavioural Analysis ... 84
8.4.3 Monte Carlo Analysis ... 87
8.4.4 Local Sensitivity Analysis ... 90
8.4.5 Optimization ... 92
8.5 User-Defined and Composed Operators / Operator Interface ... 93
8.5.1 Declaration of User-Defined Operator Dynamics ... 93
8.5.2 Undefined Results in User-Defined Operators ... 98
8.5.3 Composed Operators ... 98
8.5.4 Operator Description File <model>.odf ... 99
8.6 Undefined Results ... 101
8.7 Macros and Macro Definition File <model>.mac ... 101
8.8 Wildcard Operands &v& and &t& ... 102
8.9 Saving Results ... 103

9 VISUAL EXPERIMENT EVALUATION ... 105

10 GENERAL CONTROL, SERVICES, USER FILES, AND SETTINGS ... 107
10.1 General Configuration File <model>.cfg ... 107
10.2 Main and Auxiliary Services ... 109
10.3 SimEnv and User Include Files and Link Scripts ... 111
10.4 User Shell Scripts and Files ... 112
10.5 Built-In Names ... 115
10.6 Case Sensitivity ... 117
10.7 Nodata Representation ... 118
10.8 Environment Variables ... 118

11 STRUCTURE OF USER-DEFINED FILES, COORDINATE TRANSFORMATION FILES, VALUE LISTS ... 119
11.1 General Structure of User-Defined Files ... 119
11.2 Coordinate Transformation File ... 122
11.3 Value Lists ... 124

12 MODEL AND EXPERIMENT POST-PROCESSOR OUTPUT DATA STRUCTURES ... 127
12.1 NetCDF Model and Experiment Post-Processor Output ... 127
12.1.1 Global Attributes ... 128
12.1.2 Variable Labelling and Variable Attributes ... 128
12.2 IEEE Compliant Binary Model Output ... 130
12.3 IEEE Compliant Binary and ASCII Experiment Post-Processor Output ... 131

13 SIMENV PROSPECTS ... 133

14 REFERENCES ... 135

15 APPENDICES ... 137
15.1 Version Implementation ... 139
15.1.1 System Requirements ... 139
15.1.2 Technical Limitations ... 140
15.1.3 Linking User Models and User-Defined Operators ... 140
15.1.4 Example Models and User Files ... 141
15.1.5 Example User-Defined Operators ... 142
15.2 Examples for Model Interfaces ... 143
15.2.1 Example Implementation of the Generic Model world ... 143
15.2.2 Fortran Model ... 144
15.2.3 Fortran Model with Semi-Automated Model Interface ... 145
15.2.4 C Model ... 146
15.2.5 C++ Model ... 148
15.2.6 Python Model ... 150
15.2.7 Model Interface at Shell Script Level ... 151
15.2.8 Semi-Automated Model Interface at Shell Script Level ... 152
15.2.9 GAMS Model ... 153

15.3 Example Implementation for the Experiment Post-Processor User-Defined Operator matmul_[f | c]..... 155
15.3.1 Fortran Implementation 155
15.3.2 C Implementation 158
15.4 Example for an Experiment Post-Processor Result Import Interface 161
15.5 List of Experiment Post-Processor Built-In Operators and Operator Arguments 162
15.5.1 Experiment Post-Processor Built-In Operators (in Thematic Order) 162
15.5.2 Experiment Post-Processor Built-In Operators (in Alphabetic Order) 165
15.5.3 Character Arguments of Experiment Post-Processor Built-In Operators 168
15.5.4 Constant Arguments of Experiment Post-Processor Built-In Operators 169
15.6 Additionally Used Symbols for the Model and Operator Interface 170
15.7 Glossary 171

Tables

Tab. 1.1 Document conventions 3
Tab. 1.2 Main placeholders in this document 3
Tab. 3.1 SimEnv changes in Version 1.22 7
Tab. 3.2 User actions to upgrade to Version 1.22 8
Tab. 3.3 Limitations / problems and their workarounds 8
Tab. 3.4 Known bugs and their workarounds 9
Tab. 4.1 Statistical measures 15
Tab. 4.2 Probability density functions 16
Tab. 4.3 Local sensitivity, linearity, and symmetry measures 17
Tab. 5.1 Generic SimEnv interface functions 21
Tab. 5.2 Language suffices for SimEnv interface functions 22
Tab. 5.3 Elements of a model output description file <model>.mdf 25
Tab. 5.4 SimEnv data types 26
Tab. 5.5 Model interface functions for Fortran and C/C++ models 29
Tab. 5.6 Model interface functions for Python models 30
Tab. 5.7 Model interface functions at shell script level 32
Tab. 5.8 Elements of a GAMS description file <model>.gdf 37
Tab. 6.1 Elements of an experiment description file <model>.edf 43
Tab. 6.2 Adjustment types in experiment preparation 44
Tab. 6.3 Experiment specific elements of an edf-file for behavioural analysis 44
Tab. 6.4 Experiment specific elements of an edf-file for Monte Carlo analysis 47
Tab. 6.5 Probability density functions and their parameters 47
Tab. 6.6 Experiment specific elements of an edf-file for local sensitivity analysis 49
Tab. 6.7 Experiment specific elements of an edf-file for an optimization experiment 51
Tab. 7.1 Experiment related user shell scripts and files 59
Tab. 8.1 Classified argument restriction(s) / result description 68
Tab. 8.2 Built-in generic standard aggregation / moment operators 70
Tab. 8.3 Built-in elemental operators 70
Tab. 8.4 Built-in basic and trigonometric operators 71
Tab. 8.5 Built-in standard aggregation / moment operators without suffix 73
Tab. 8.6 Built-in standard aggregation / moment operators with suffix _n 73
Tab. 8.7 Built-in standard aggregation / moment operators with suffix _l 74
Tab. 8.8 Built-in advanced operators 77
Tab. 8.9 Multi-run standard aggregation / moment operators 83
Tab. 8.10 Experiment specific operator for behavioural analysis 84
Tab. 8.11 Syntax of the filter argument 1 for operator behav 85
Tab. 8.12 Experiment specific operators for Monte Carlo analysis 88
Tab. 8.13 Experiment specific operators for local sensitivity analysis 90
Tab. 8.14 Syntax of the filter argument 1 for local sensitivity operators 90
Tab. 8.15 Operator interface functions for the declarative and computational part 93
Tab. 8.16 Operator interface functions to get and put structural information 96
Tab. 8.17 Operator interface functions to get / check / put arguments and results 97
Tab. 8.18 Elements of an operator description file <model>.odf 100
Tab. 8.19 Elements of a macro description file <model>.mac 101

Tab. 10.1 Elements of a general configuration file <model>.cfg 107
Tab. 10.2 Default values for the general configuration file 109
Tab. 10.3 SimEnv services 110
Tab. 10.4 SimEnv and user include files and link scripts 111
Tab. 10.5 Contents of $SE_HOME/simenv_mod_[f | c].inc 111
Tab. 10.6 User files and shell scripts to perform any SimEnv service 112
Tab. 10.7 Files generated during performance of SimEnv services 114
Tab. 10.8 Built-in model output variables 115
Tab. 10.9 Built-in coordinates 116
Tab. 10.10 Built-in shell script variables in <model>.run 116
Tab. 10.11 Case sensitivity of SimEnv entities 117
Tab. 10.12 Data type related nodata values 118
Tab. 10.13 Environment variables 118
Tab. 11.1 User-defined files with general structure 119
Tab. 11.2 Constraints in user-defined files 120
Tab. 11.3 Reserved names and file names in user-defined files 120
Tab. 11.4 Line types in user-defined files 121
Tab. 11.5 Elements of a coordinate transformation file 122
Tab. 11.6 Syntax rules for value lists 124
Tab. 12.1 NetCDF data types 127
Tab. 12.2 Additional global NetCDF attributes 128
Tab. 12.3 Variable NetCDF attributes 128
Tab. 12.4 Variable NetCDF attributes for visualization 129
Tab. 15.1 SimEnv installation under Unix 139
Tab. 15.2 System requirements 139
Tab. 15.3 Current SimEnv technical limitations 140
Tab. 15.4 Implemented example models for the current version 141
Tab. 15.5 Implemented model and operator related user files for the current version 142
Tab. 15.6 Available user-defined operators 142
Tab. 15.7 Targets of the generic model world 143
Tab. 15.8 Experiment post-processor built-in operators (in thematic order) 164
Tab. 15.9 Experiment post-processor built-in operators (in alphabetical order) 167
Tab. 15.10 Character arguments of experiment post-processor built-in operators 168
Tab. 15.11 Constant arguments of experiment post-processor built-in operators 169
Tab. 15.12 Additionally used symbols for the model interface 170
Tab. 15.13 Additionally used symbols for the operator interface 170

Figures

Fig. 0.1 SimEnv system design 2
Fig. 4.1 Target space 12
Fig. 4.2 Sample for a behavioural analysis 13
Fig. 4.3 Behavioural analysis: Scanning multi-dimensional target spaces 14
Fig. 4.4 Sample for a Monte Carlo analysis 14
Fig. 4.5 Sample for a local sensitivity analysis 17
Fig. 4.6 Part of a sample for an optimization experiment, generated during the experiment 19
Fig. 5.1 Conceptual scheme of the model interface for C/C++/Fortran/Python 23
Fig. 5.2 Grid types 24
Fig. 5.3 Model output variable definition: Grid assignment 27
Fig. 6.1 Monte Carlo analysis: Latin hypercube sampling 48
Fig. 7.1 Flowcharts for performing simenv.run and simenv.rst 60
Fig. 10.1 SimEnv user shell scripts and files 115

Examples

Example 1.1 General example layout in the User Guide ... 4
Example 5.1 Model output description file <model>.mdf ... 27
Example 5.2 Addressing target names and values for the model interface at shell script level ... 33
Example 5.3 Model output description file for a GAMS model ... 36
Example 5.4 GAMS description file <model>.gdf ... 37
Example 5.5 GAMS description file for coupled GAMS models ... 38
Example 5.6 Shell script <model>.run for a parallel model ... 41
Example 6.1 Experiment description file <model>.edf for behavioural analysis ... 46
Example 6.2 Experiment description file <model>.edf for Monte Carlo analysis ... 49
Example 6.3 Experiment description file <model>.edf for local sensitivity analysis ... 50
Example 6.4 Experiment description file <model>.edf for an optimization experiment ... 52
Example 7.1 Shell script <model>.run to wrap the user model ... 55
Example 7.2 Shell script <model>.ini for user-model specific experiment preparation ... 55
Example 7.3 Shell script <model>.end for user-model specific experiment wrap-up ... 55
Example 7.4 Shell script <model>.rst to prepare model performance during experiment restart ... 57
Example 8.1 Addressing results in experiment post-processing ... 64
Example 8.2 Addressing model output variables in experiment post-processing ... 66
Example 8.3 Checking rules for coordinates ... 69
Example 8.4 Experiment post-processing operator get_data and coordinate transformation file ... 78
Example 8.5 Experiment post-processing with advanced operators ... 82
Example 8.6 Experiment post-processing operator behav for behavioural analysis ... 87
Example 8.7 Experiment post-processing operators for Monte Carlo analysis ... 89
Example 8.8 Experiment post-processing operators for local sensitivity analysis ... 91
Example 8.9 Composed operators ... 99
Example 8.10 Operator description file <model>.odf ... 100
Example 8.11 User-defined macro definition file <model>.mac ... 102
Example 8.12 Experiment post-processing with wildcard operands ... 103
Example 10.1 User-defined general configuration file <model>.cfg ... 109
Example 11.1 Structure of a user-defined file ... 121
Example 11.2 Coordinate transformations by a transformation file ... 124
Example 11.3 Examples of value lists ... 125
Example 12.1 IEEE compliant model output data structure ... 130
Example 15.1 Model interface for Fortran models - model world_f.f ... 144
Example 15.2 Semi-automated model interface for Fortran models - model world_f_auto.f ... 145
Example 15.3 Model interface for C models – model world_c.c ... 147
Example 15.4 Model interface for C++ models – model world_cpp.cpp ... 149
Example 15.5 Model interface for Python models – model world_py.py ... 150
Example 15.6 Model interface at shell script level – model shell script world_sh.run ... 151
Example 15.7 Semi-automated model interface at shell script level – model shell script world_sh_auto.run ... 152
Example 15.8 Model interface for GAMS models – model gams_model.gms ... 154
Example 15.9 Experiment post-processor user-defined operator module – operator matmul_f ... 157
Example 15.10 Experiment post-processor user-defined operator module – operator matmul_c ... 160
Example 15.11 ASCII compliant experiment post-processor result import interface ... 161

That is what we meant by science. That both question and answer are tied up with uncertainty, and that they are painful. But that there is no way around them. And that you hide nothing; instead, everything is brougth out into the open.

Peter Høeg, Borderliners
Mc Clelland-Bantam, Toronto, 1995, p. 19

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, optionally supplemented by user-defined and composed operators and applies operator chains on model output and reference data. Result output functions generated during post-processing can be evaluated within SimEnv with advanced visualization techniques.

Simulation is one of the cornerstones for research. 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 & 2004) and scenario analyses. Both research foci require complex simulation experiments for model inspection, validation and control design without changing the model in general.

SimEnv (Flechsig *et.al*, 2005) 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 *t* 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 *M*. 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 variable's *z*) local sensitivity to targets *t*. Is performed by finite difference derivative approximations from *M*.
  For numerical validation purposes, model analysis, sub-model sensitivity.
- Optimization
  Iterative determination of optimal targets *t* for a cost functions derived from *z* by a simulated annealing method.
  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 interface functions into Fortran-, C/C++-, Python- or GAMS-model source code for target 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 / distributed architectures to distribute work load of the single runs of the experiment.
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 operator chains. SimEnv supplies built-in operators and enables specification of user-defined and composed operators.

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

SimEnv key features:

- Available at Unix and Linux platforms
- 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 (parameters, initial values, drivers, ...) to experiment with and a strategy how to sample the target space have to be specified.
- Simple model interface to the simulation environment:
  Model interface functions allow mainly to re-adjust an experiment target and to output model results for later experiment post-processing. Model interfacing and finally communication between the model and SimEnv can be done at the model language level by incorporating interface function calls into model source code (C/C++, Fortran and Python: “include per experiment target and per model output variable one additional SimEnv function call into the source code”) or can be done at the shell script level. Additionally, there is a special interface for GAMS models.
- Support of distributed models:
  Independently on the kind distributed model components are interfaced to SimEnv and among each other the total model can be run within SimEnv.
- Parallelization of the experiment:
  This is a prerequisite for a lot of simulation tasks.
- Operator-based experiment post-processing:
  Chains of built-in, user-defined and composed operators enable interactive experiment post-processing based on experiment model output and reference data including general purpose and experiment specific operators. There is a simple interface to write user-defined and to derive composed 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.

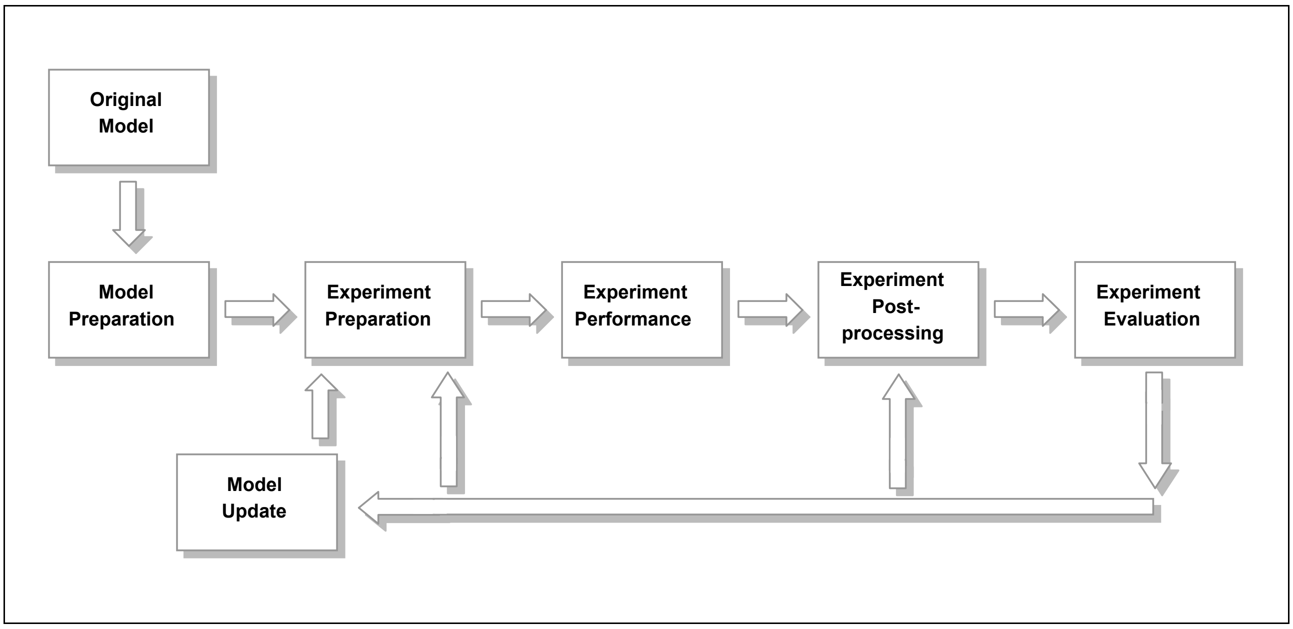


Fig. 0.1 *SimEnv system design*

1 About this Document

In this chapter document conventions are explained. Within the whole document one generic 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
‘ ... ‘	single quotation marks enclose a keyword or sub-keyword from user-defined files
“ ... “	double quotation marks enclose the string-value of a sub-keyword from user-defined files
<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
<directory>	path to a file directory
<file_name>	name of a data file
<GAMS_model>	name of a GAMS model to start a SimEnv service with
<int_val>	integer value
<model>	model name to start a SimEnv service with
<real_val>	real value in integer, fixed point (e.g., -1.234) or floating point (scientific) (e.g., -0.1234e+1) notation
<res>	integer experiment post-processor output file number 1, 2, ..., 99
<res_char>	character experiment 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 and related files
<string>	any string
<target_def_val>	default value of a target as defined in <model>.edf
<target_name>	name of a target to experiment with as defined in <model>.edf
<val_list>	list of values in explicit or implicit notation according to Tab. 11.6
For post-processor operator descriptions only	
arg	general numerical argument (operand)
int_arg	integer constant argument (operand) ≥ 0
real_arg	real (float) constant argument (operand)
char_arg	character argument (operand), enclosed in single quotation marks

Tab. 1.2 *Main placeholders in this document*

1.2 Example Layout

All examples in this document but that for the GAMS model (see Section 5.5 on page 30) refer to a hypothetical global simulation **model world**. 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.

The model world is assumed to map lateral and vertical (level) fluxes and demands that's why for computing state variables for the whole globe.

The model world is a generic model. Model implementation in several programming languages results in models world_<lng> where <lng> is an identifier for the programming language (and the lateral model resolution).
In the model gridcell_f state variables are calculated for one grid cell (one single latitude - longitude constellation) without consideration of lateral fluxes.

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 84°N and 60°S latitude at land masses, i. e., without Antarctic)	lat x lon x time	float
atmo_g (not for model gridcell_f)	aggregated global state derived from atmo for level 1	time	int
bios_g (not for model gridcell_f)	aggregated global state derived from bios	-	int

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

With this SimEnv release the following model implementations are distributed:

Model (“auto” in name = semi-automated model interface)	**Model interface example for language <lng>**	**Resolution**		
		lateral: lat x lon	**vertical: number of levels**	**temporal: number 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	Shell script level	4 x 4	4: 1, 7, 11, 16	20
world_f_1x1	Fortran	1 x 1	16: 1 - 16	20
world_f_05x05	Fortran	0.5 x 0.5	16: 1 - 16	20
world_f_auto	Fortran	4 x 4	4: 1, 7, 11, 16	20
world_sh_auto	Shell script level	4 x 4	4: 1, 7, 11, 16	20
gridcell_f	Fortran	without, implicitly by experiment as 4 x 4	4: 1, 7, 11, 16	20

Examples are generally placed in grey-shaded boxes. Examples that are available from the example directory $SE_HOME/../examples of SimEnv are marked as such in the lower right corner of an example box. To copy files from this directory use the SimEnv service simenv.cpy (see Tab. 10.3).

Example 1.1 *General example layout in the User Guide*

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 files are essential to use the simulation environment.

- SimEnv is implemented under AIX-Unix at IBM's RS6000 and compatibles and SUSE-Linux at Intel-based platforms and compatibles. For detailed system requirements check Tab. 15.2 on page 139.
- Include into the file $HOME/.profile

```
export SE_HOME=/usr/local/simenv/bin
export PATH=$SE_HOME:$PATH
```

  to set the SimEnv home directory and execute at the operating system prompt

```
. $HOME/.profile
```

- Change to a directory with full access permissions. This is the SimEnv current workspace.
- Start

```
simenv.hlp
```

  to acquire basic information on how to use SimEnv.
- Select a model implementation language <lng> to check SimEnv with the model world_<lng> from Example 1.1 on page 4:

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

  For a GAMS model example check Section 5.7 on page 33.
- Start

```
simenv.cpy    world_<lng>
```

  to copy the model world_<lng> model and experiment related files to the current workspace.
- Copy the file world.edf_c to world_<lng>.edf
- Check

		for
• The SimEnv configuration file	**world_<lng>.cfg**	general SimEnv configurations
• The model output description file	**world_<lng>.mdf**	available model output variables
• The model	**world_<lng>.<lng>**	implementation of the model
• The model wrap 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 result sequence

- Start

```
simenv.cpl    world_<lng>   -1   world.post_c
```

  to run a complete SimEnv session:
  - Model and experiment related files will be checked
  - The experiment will be prepared
  - The experiment will be performed (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 result and finish post-processing by entering a single <return>
  - Visualization of post-processed results will be started **(*)**
  - Model or result output files will be dumped

  or

- Start

  ```
  simenv.chk     world_<lng>
  ```

  to check model and experiment relate files.
- Start

  ```
  simenv.run     world_<lng>
  ```

  to prepare and perform a simulation experiment (select the login machine on request).
- Start

  ```
  simenv.res     world_<lng>   { new   { <run> } }
  ```

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

  ```
  simenv.vis     world_<lng>   { [ latest | <res_char> ] }
  ```

  to visualize output from the latest post-processing session world_<lng>.res<res_char>.nc or that with number <res_char> with the highest two-digit number <res_char>.
- Start

  ```
  simenv.dmp     world_<lng>   mod   |  more
  simenv.dmp     world_<lng>   res   |  more
  ```

  to dump a SimEnv model or post-processor output file.

- Check in the current workspace the

model interface	log-file	world_<lng>.mlog
native model terminal output	log-file	world_<lng>.nlog
experiment performance	log-file	world_<lng>.elog.

- Start

  ```
  simenv.cln     world_<lng>
  ```

  to wrap up a simulation experiment.
- Get the usage of any SimEnv service by entering the service 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 permission for the visualization server check in Section 10.2 on page 109 the SimEnv service

```
simenv.key     <user_name>
```

3 Version 1.22

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

3.1 What is New?

Type	Check / see	On page	Description
update / new	Section 6.1	43	Experiment preparation and performance: Unification and simplification of target adjustments. Target adjustment values are sampled to <model>.edf_adj as specified in <model>.edf and are modified within simenv_get_* by the target default value in a unique manner according to the target type. Special rules for Monte Carlo and local sensitivity analysis are unnecessary furtheron. Introduction of an additional adjustment type = relative: adjusted target value = (1. + <adj_val>) * <target_def_value>
new	Section 7.3	55	Experiment performance controlled by LoadL: Experiment single runs now can be distributed to all processors of a job class, exploiting in the course of the experiment the free processors of the class in an optimal manner.
new	Section 8.8	102	Wildcard operands in the experiment post-processor: Use wildcard operands to perform a result expression for all model output variables and/or experiment targets.
update	Section 7.2 Section 7.3	54 55	SimEnv runs now in the Bourne shell sh: Update the first line in <model>.run and <model>.rst accordingly
update	Section 10.8	118	The whole operating system environment necessary for SimEnv is set within SimEnv. The dot script simenv.env is now dispensable. Exception (include in $HOME/.profile): export SE_HOME=/usr/local/simenv/bin (default) export PATH=$SE_HOME:$PATH (optional)
update	Tab. 11.3	120	Operating system environment variables now can be used when specifying directories in user-defined files and as arguments in post-processing operators
update	Tab. 10.4 Section 7.3 Section 6.5.1	111 55 51	Files renamed: in $SE_HOME: simenv_mod_inc.[f | c] to simenv_mod_[f | c].inc simenv_opr_inc.[f | c] to simenv_opr_[f | c].inc <model>_inc.[f | c | py | sh] to <model>_[f | c | py | sh].inc in the current workspace: simenv.[jcf_par | jcf_seq | opt_opt] to <model>.*
			Bug fixes

Tab. 3.1 *SimEnv changes in Version 1.22*

Upgrade type	Upgrade action
mandatory	Update source code of models (Fortran/C/C++) and user-defined operators: rename include files
mandatory	Update <model>.edf for Monte Carlo and local sensitivity analyses
mandatory	Rename optional job control files and optimization options file
mandatory	Update <model>.run and <model>.rst: change shell
mandatory	Update $HOME/.profile
mandatory	Re-link models interfaced to SimEnv and user-defined operators

Tab. 3.2 *User actions to upgrade to Version 1.22*

3.2 Limitations / Problems and Their Workarounds

Where Limitation / Problem Workaround	Description
Where	Overall
Limitation	Current SimEnv technical limitations as specified in Tab. 15.3 on page 140
Workaround	None
Where	Overall but visual result evaluation
Limitation	Without graphical user interface
Workaround	None
Where	Experiment performance: Experiment type optimization
Limitation	Can not be performed in parallel mode
Workaround	Perform optimization experiment in sequential mode
Where	Experiment performance: Experiment type optimization
Limitation	The initial seed for the optimization technique is fixed. That's why the algorithm results for the same optimization problem always in the same sampled sequence in the target space
Workaround	None
Where	Experiment performance: Model output to NetCDF
Problem	Check on undefined model output results in noticably additional CPU-time consumption. Example: to check 8 Mill of real*8 values takes per single run additionally 80 sec for single nc-file model output and 200 sec for common nc-file output.
Workaround	Specify in <model>.cfg for sub-keyword 'message_level' value = "error"
Where	Experiment post-processing: Optional specification / automated identification of result description and result unit
Limitation	Not stored to NetCDF result output
Workaround	Specify IEEE or ASCII result output instead

Tab. 3.3 *Limitations / problems and their workarounds*

3.3 Known Bugs and Their Workarounds

Where Bug Workaround	**Description**
Where	Experiment restart Model output to a common NetCDF file for the whole experiment
Bug	Model output is not stored
Workaround	Specify IEEE model output or single NetCDF file output in <model>.cfg
Where	Experiment performance: Model output to NetCDF of distributed models (structure = distributed in <model>.cfg)
Bug	May not store all model output
Workaround	Specify IEEE model output in <model>.cfg
Where	Experiment post-processing: Result output to NetCDF
Bug	simenv.res <model> append does not work
Workaround	Specify result output to IEEE or ASCII instead
Where	Experiment post-processing: Behavioural analysis / result output to NetCDF
Bug	When applying the operator behav non-monotonic and monotonously decreasing target adjustments are transferred to the NetCDF output file in a wrong manner.
Workaround	Specify only monotonously increasing target adjustments in <model>.edf or specify IEEE and/or ASCII post-processor output in <model>.cfg

Tab. 3.4 *Known bugs and their workarounds*

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

$$M: \quad Z(t) = ST\ (\ Z(t-\Delta t)\ ,\ldots,\ Z(t-k^*\Delta t)\ ,\ P\ ,\ X(t)\ ,\ Z_0\ ,\ B\ )$$

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
	k	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 k 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 / triggers 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 pre-determined in the model by assigning values of a technical / trigger parameter p_{tech} to applying for example alternative model structures, sub-structures, processes formulations, resolutions, which are triggered 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 SimEnv all the four model components parameters, drivers, initial values and boundary values are lumped together and the term **target**[1] stands as a placeholder for them. All targets form the target set T:

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

and

$$Z = ST(\mathrm{T}).$$

In the following,

$$\mathrm{T}_k = (t_1 ,..., t_k) \qquad k > 0$$

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

$$\mathrm{T}_{k,n} = \begin{pmatrix} t_{11} & ... & t_{1n} \\ ... & & ... \\ t_{k1} & ... & t_{kn} \end{pmatrix} = (\check{\mathrm{T}}_1 ,..., \check{\mathrm{T}}_n) \qquad k > 0,\ n > 0$$

stands for a numerical sample for T_k of size n and finally for k*n values representing in any sense the sample space T_k.
In the set of all samples $\mathrm{T}_{k,1}$ $T_{k,1}$ is 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 holds:

$$\{ Z \}_n = \{ ST(\check{\mathrm{T}}_1) ,..., ST(\check{\mathrm{T}}_n) \}.$$

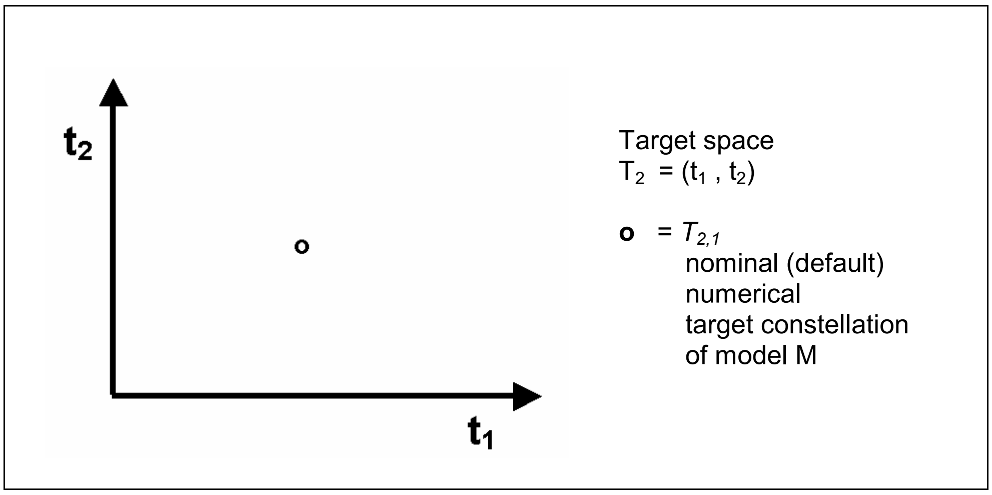


Fig. 4.1 *Target space*

SimEnv supports different sampling strategies and performance of multi-run experiments where m targets are readjusted numerically for each of n single simulation runs. Central goal is to study dependency of the model dynamics on target adjustments. For simulation purposes in SimEnv experimentation with the model *M* over $\mathrm{T}_{k,n}$ is based on the assumption that dynamics of *M* for each representative from the sample is independent 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 $\mathrm{T}_{k,n}$.

SimEnv experiment types differ in the way the sample space T_k is sampled to get $\mathrm{T}_{k,n}$. 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.

[1] The term target was selected as an analogue to experimentation with real systems: Often a target is under investigation to study the change in the real system when the state of the target is modified by the experimentor. Often used synonyms for “target” are “input” and “factor”.

4.2 Behavioural Analysis

Behavioural analysis uses a deterministic strategy to sample T_k. It is the inspection of the model in the target space T_k 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

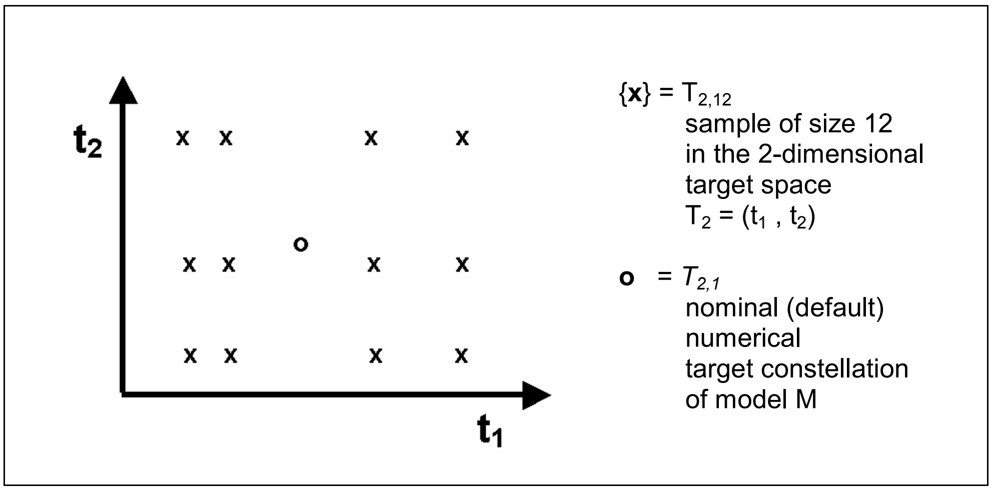


Fig. 4.2 *Sample for a behavioural analysis*

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_k 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). Experiment post-processing can resolve the scanning method again and output 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 ● in Fig. 4.3 correspond to an cross **x** in Fig. 4.2 and represents a sample point in the target space and finally a single model run of the experiment.

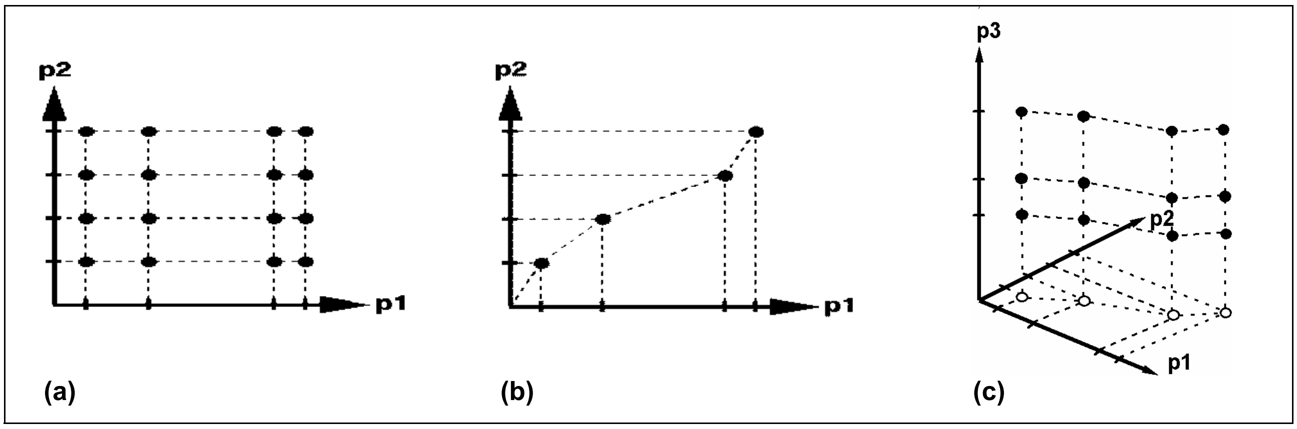


Fig. 4.3 *Behavioural analysis: Scanning multi-dimensional target spaces*

4.3 Monte Carlo Analysis

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

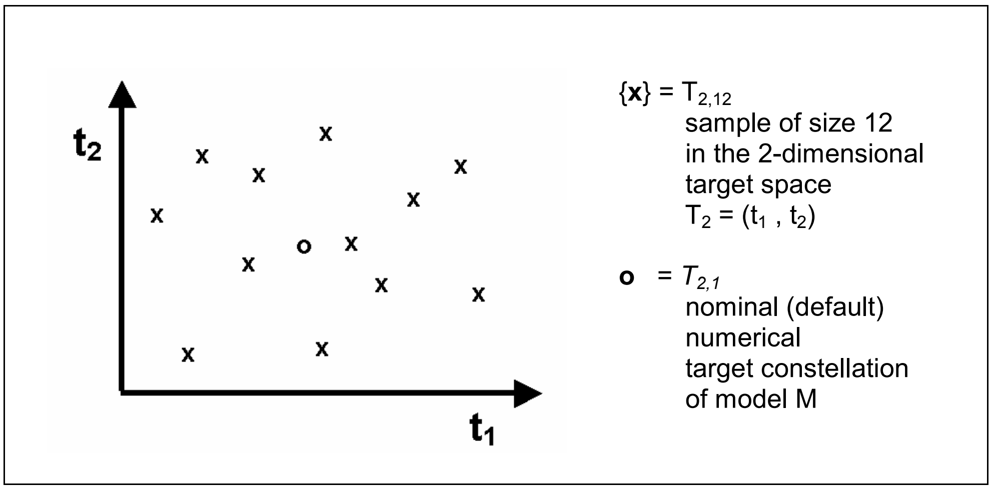


Fig. 4.4 *Sample for a Monte Carlo analysis*

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

$$M^{(m)}\{z\} = \int\limits_{T_k}\cdots\int z(T_k)^m * \mathrm{pdf}(T_k)\, dT_k$$

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

By interpreting the probability density function $\mathrm{pdf}(T_k)$ as the error distribution in the target space T_k 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 $\mathrm{pdf}(T_k)$ can be decomposed into independent probability density functions pdf_i for all targets t_i of T_k:

$$\mathrm{pdf}(T_k) = \prod_{i=1}^{k} \mathrm{pdf}_i(t_i)$$

and the k-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 ($1 \le i \le k$, $1 \le j \le n$) 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 an experiment post-processing session from a Monte Carlo analysis with n simulation runs resulting in n realizations z_1 ,..., z_n of the model's state variables z, z1 and z2:

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

Tab. 4.1 *Statistical measures*

(*)*: indices for sums* Σ, *products* Π *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)	$\mathrm{pdf}(x) = \frac{1}{b-a}$	if $x \in [a,b]$	a b	lower boundary upper boundary > a
		pdf(x) = 0	otherwise	it is:	mean = (a+b) / 2 standard deviation = $\sqrt{(b-a)^2 / 12}$
normal	N(μ,σ^2)	$\mathrm{pdf}(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$		μ σ	mean standard deviation > 0
lognormal	L(μ,σ^2)	$\mathrm{pdf}(x) = \frac{1}{x\sigma\sqrt{2\pi}} \exp\left(-\frac{(\ln x-\mu)^2}{2\sigma^2}\right)$	if x > 0	μ σ	 > 0
		pdf(x) = 0	otherwise	it is:	ln(x) ~ N(μ,σ^2)
exponential	E(μ)	$\mathrm{pdf}(x) = \frac{1}{\mu} \exp\left(-\frac{x}{\mu}\right)$	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 $T_{k,1}$ of the model M. For each target t_i from the nominal target constallation $T_{k,1}$ and each ε_j from the ε-neighbourhoods (ε_1 ,…, ε_m) two members (t_1 ,..., t_{i-1} , $t_i \pm \varepsilon_j$, t_{i+1} ,..., t_k) of the resulting sample are generated. The sample size n is given by 2*m*k. Running the model for 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 $\delta z/\delta t$. In the numerical simulation of complex systems a finite sensitivity function is preferred, because it can be obtained without model enlargements or re-formulations. It is a linear approximations of the classical model sensitivity measure (Wierzbicki, 1984). Contrary to a global sensitivity analysis a local one covers the model's sensitivity around the nominal target constellation.

Local sensitivity measures as well as measures which reflect model output linearity and/or symmetry nearby $T_{k,1}$ 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 experiment post-processing is based on finite sensitivity, linearity, and symmetry measures, which are defined as in Tab. 4.3.

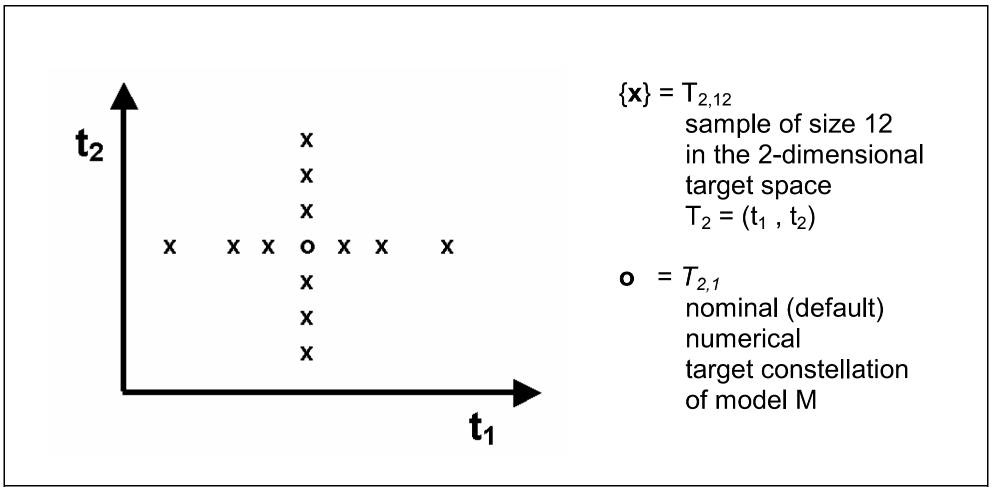


Fig. 4.5 *Sample for a local sensitivity analysis*

Local measure	Definition	
	Absolute measure	**Relative measure**
sensitivity measure	sens_abs(z,±ε) $= \dfrac{z(t \pm \varepsilon) - z(t)}{\pm \varepsilon}$	sens_rel(z,±ε) = sens_abs(z,±ε) $\dfrac{t}{z(t)}$
linearity measure	lin_abs(z,ε) $= \dfrac{(z(t+\varepsilon) - z(t)) + (z(t-\varepsilon) - z(t))}{\varepsilon}$	lin_rel(z,ε) = lin_abs(z,ε) $\dfrac{t}{z(t)}$
symmetry measure	sym_abs(z,ε) $= \dfrac{z(t+\varepsilon) - z(t-\varepsilon)}{\varepsilon}$	sym_rel(z,ε) = sym_abs(z,ε) $\dfrac{t}{z(t)}$

Tab. 4.3 *Local sensitivity, linearity, and symmetry measures*
for a selected target t from $T_{k,1}$ and a selected ε from (ε_1 ,…, ε_m)

Accordingly, local measures of the model with respect to a target are always expressed as a measure of a model's state variable z, usually at a selected time step within a surrounding neighborhood ε of a target value t. That is why the conclusions drawn from a local sensitivity analysis are only valid locally at $T_{k,1}$ with respect to the whole target space T_k. Additionally, local measures only describe the influence of one target t_i from the whole vector T_k on the model's dynamics.

As stated above, the sensitivity measures reflect the classical sensitivity functions in a neighborhood of $T_{k,1}$. The larger the absolute value of the measure the higher is the influence of an incremental change of the target t on the model output z. The linearity measures map the linear behaviour of z nearby $T_{k,1}$. If the linear measure is zero z shows a linear behaviour with respect to t. The symmetry measures measures map the symmetric behaviour of the z nearby $T_{k,1}$. If the symmetry measure is zero z shows a symmetric behaviour with respect to t. The larger the absolute values of the latter two measures the higher is the nonlinear / non-symmetric behaviour of z with respect to t.

The absolute measures are best suited to compare the influence of different targets {t} on the same state variable z while due to their normalization factor the relative measures enable comparison of the influence of one target t on different state variables {z}.

From the local measures of table Tab. 4.3 additional measures can be derived on demand, e.g., abs(sym_abs(z, ε)).

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

4.5 Optimization

The optimization experiment in SimEnv uses a stochastic strategy to sample T_k. It is the only experiment type where the sample is generated during experiment performance and not at experiment preparation. The general approach of optimization is to find the global minimum of a cost function (synonym: objective function)

$$F(Z) = F(ST(T_k))$$

that depends on model's state variables Z and consequently on the experiment targets $T_k = (t_1 ,..., t_k)$:

$$\begin{array}{ll} \text{minimize} & F(t_1 ,..., t_k) \\ \text{subject to} & t_{i\,min} \le t_i \le t_{i\,max} \quad \text{for } i = 1 ,..., k \end{array}$$

Often, F represents a distance measure in a specific metric between selected model state variables and reference data (measurement values of the real system or simulation results from an other model). Consequently, optimization can be used for model validation and control design to find optimal values of model targets in such a way that model state variables are close to reference data. In SimEnv the cost function is specified in experiment preparation as a single run result formed from model output (and reference data) where an operator chain is applied on (check Section 6.5 and Chapter 8). The value of the cost function is calculated directly after the current single run has been performed.

SimEnv uses a gradient free optimization approach that is called **“Simulated Annealing”** and is a generalization of a Monte Carlo method for examining the state equations of n-body systems. The concept is based on the manner in which metals recrystalize in the process of annealing. In an annealing process a melt, initially at high temperature Temp and disordered, is slowly cooled so that the system at any time is approximately in thermodynamic equilibrium. As cooling proceeds, the system becomes more ordered and approaches a "frozen" ground state at Temp = 0. Hence the process can be thought of as an adiabatic approach to the lowest energy state E. If the initial temperature of the system is too low or cooling is done insufficiently slowly the system may become quenched forming defects or freezing out in metastable states (i.e. trapped in a local minimum energy state).

The annealing scheme is that an initial state of a thermodynamic system is chosen at energy E and temperature Temp, holding Temp constant the initial configuration is perturbed and the change in energy dE is computed. If the change in energy is negative or zero the new configuration is accepted. If the change in energy is positive it is accepted with a probability given by

$$p = \exp(-dE/(k_B{*}Temp))$$

where k_B denotes the Boltzmann constant. This process is then repeated sufficient times to give good sampling statistics for the current temperature, and then the temperature is decremented and the entire process repeated until a frozen state is achieved at Temp = 0.

By analogy the generalization of this Monte Carlo approach to optimization problems is straight forward:

- The current state of the thermodynamic system is analogous to the current solution to the optimization problem
- The energy equation for the thermodynamic system is analogous to the objective function F, and
- The ground state at Temp = 0 is analogous to the global minimum of F.

The major difficulty (art) in implementation of a simulated annealing algorithm is that there is no obvious analogy for the temperature Temp with respect to a free parameter in the optimization problem. Furthermore, avoidance of entrainment in local minima (quenching) is dependent on the "annealing schedule", that is, the choice of initial temperature, how many iterations are performed at each temperature, and how much the temperature is decremented at each step as cooling proceeds (after Gray *et al*., 1997). Ideally, when local optimization methods are trapped in a poor local minimum, simulated annealing can ‘climb’ out.

The algorithm applied in SimEnv is a very fast simulated re-annealing method, named Adaptive Simulated Annealing ASA (Ingber 2004, Ingber 1989 and Ingber 1996). For the above stated probability p the term k_B * Temp is chosen as

$$k_B * Temp = Temp_0 * \exp(-c*k^{1/m})$$

where k is the annealing time.

The ASA schedule is much faster than Boltzmann annealing, where k_B * Temp = $Temp_0$ /ln(k) and faster than fast Cauchy annealing, where k_B * Temp= $Temp_0$ / k. With the ASA method the global minimum of a nonlinear non-convex cost function F over an m-dimensional bounded target space T_k is determined.

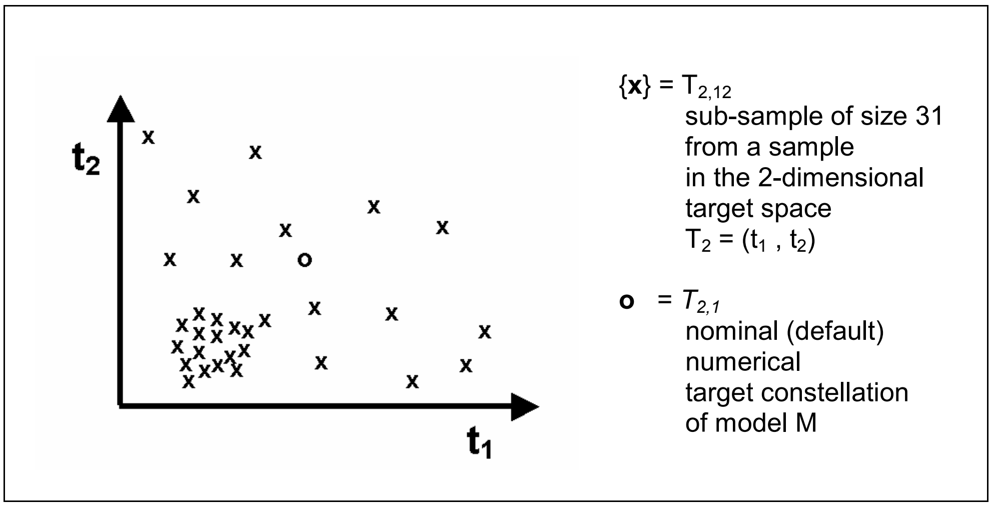


Fig. 4.6 *Part of a sample for an optimization experiment, generated during the experiment*

5 Model Interface

To use any model within SimEnv it has to be interfaced 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 potentially processed during experiment post-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 output variables under investigation from the model to the simulation environment for later experiment post-processing. Interfacing is supported at the programming language level for C/C++, Fortran, Python, and GAMS programming languages, the model is implemented in and at shell script level.

5.1 General Approach

SimEnv model interface has to supply a link between the simulation environment and the model and has to address two aspects:
For each single run from the run ensemble

- All numerical adjustments of experiment targets as defined in the experiment description file <model>.edf (check Section 6.1) have to be associated to the corresponding model entities (parameters, initial or boundary values, drivers) and these entities have to be modified numerically in the model according to the specified adjustments.
- All model output variables as defined in the model output description file <model>.mdf (check Section 5.3) have to be associated to the corresponding model entities (in general, model state variables) and these entities have to be output to SimEnv data structures during the performance of the model.

Realisation of this general approach is based on minimal source code manipulation of the model. SimEnv supplies a library with a set of simple functions to interface the model to the simulation environment. Generally speaking,

- Every experiment target and
- Every model output variable

demand one additional SimEnv function call in the model source code. According to Tab. 5.1 model interface functions are generic.

Function name	Description
simenv_ini_<lng>	open model coupling interface
simenv_get_<lng>	associate a model source code entity with an experiment target (parameter / initial value / boundary value) from <model>.edf and get the target adjustment
simenv_get_run_<lng>	get the current single run number of the run ensemble
simenv_put_<lng>	associate a model source code entity with a model output variable from <model>.mdf and output it to SimEnv data structures
simenv_slice_<lng>	enable slicing, i.e., a repetitively partial output of model output variables.
simenv_end_<lng>	close model coupling interface

Tab. 5.1 *Generic SimEnv interface functions*
(for <lng> check Tab. 5.2*)*

The function simenv_slice_<lng> announces output of a slice of the data of a defined model output variable. This is good for models with multi-dimensional variables where at least one dimension is omitted in the state variable declaration in the model the source code 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 simenv_slice_<lng>-function ensures that model output over the omitted dimension can be handled in experiment post-processing in common.

Fig. 5.1 shows the conceptual scheme for the SimEnv interface for a Fortran model.

The alignment of the contents of the SimEnv description files and the used SimEnv model interface functions in the model source code is dominated by the description files: These files determine the experiment and the model source code is expected to be well adapted. Nevertheless, this approach is implemented in a flexible manner:

- Function calls in the source code where an experiment target from <model>.edf and/or a model output variable from <model>.mdf is not associated with are handled during the model performance in such a way that the targets are unadjusted and/or the model output variable is not output. This enables adaption of the model source code for a number of potential experiment targets and model outputs where only a subset of these targets is under consideration in special experiments and/or requested for model output.
- *Vice versa*, model entities that are requested by the corresponding experiment and/or model output description file for target adjustments and/or model output and where the corresponding SimEnv functions in the model source code are missing are identified as such.

A regular matching between the model output description file and the used SimEnv interface functions in the model source code as well as the above exceptions are reported to the interface log-file <model>.mlog (check Tab. 10.7).

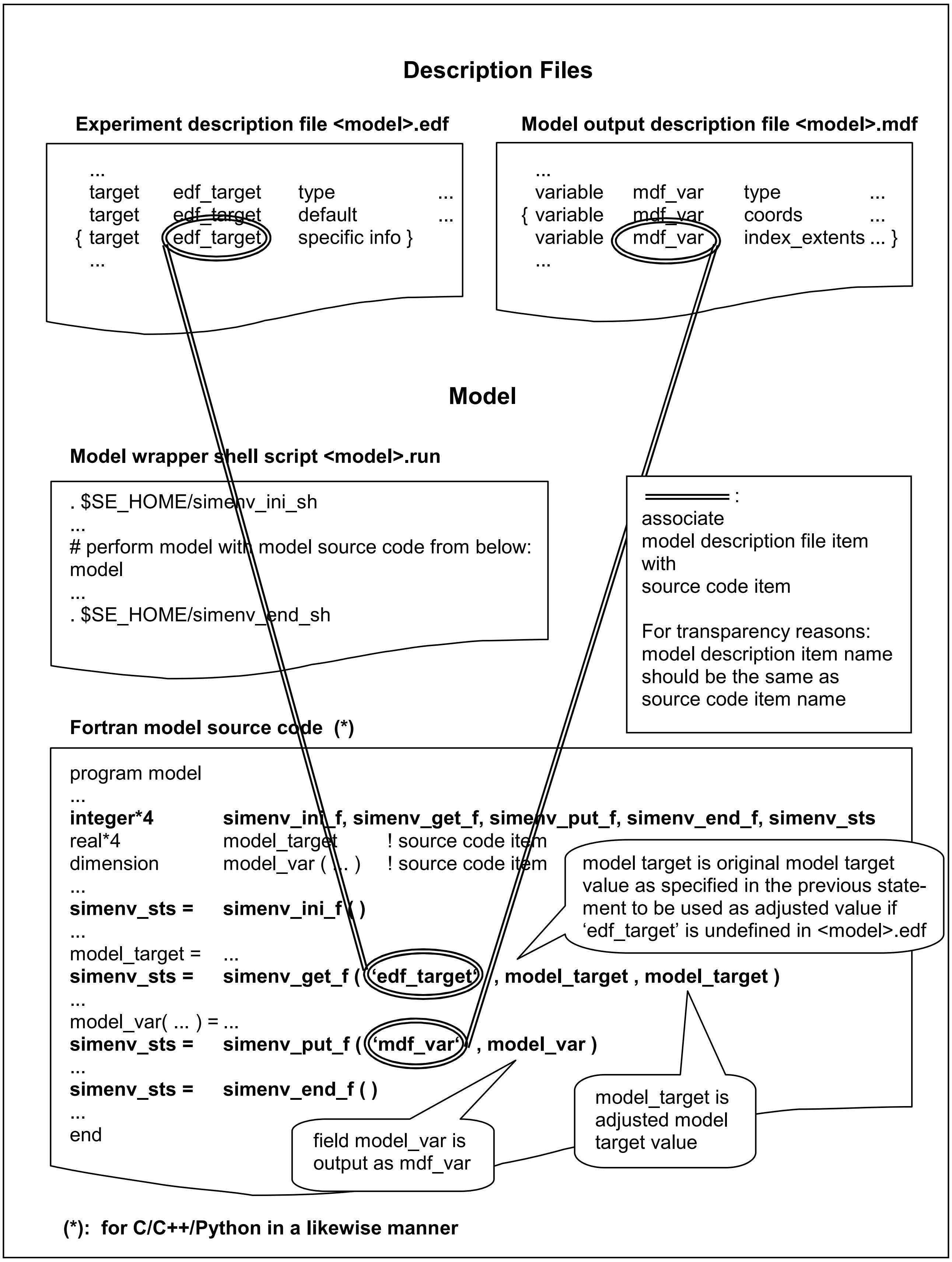
Native model output does not influence performance of the model in SimEnv and there is no necessity to disable this output for SimEnv. The user only has to ensure that for a experiment control by the load leveler LoadL the outputs of different single runs do not conflict with each other. Normally, this can be ensured by performing each single run in a special run-related sub-directory (check Example 15.6). Native user model output to the terminal is redirected during the experiment to the log-file <model>.nlog.

For running an interfaced model outside SimEnv there are dummy SimEnv libraries to link / run the model with. They ensure the same model dynamics as before interfacing the model to SimEnv (check Section 5.10).

Currently, there are SimEnv interfaces for Fortran, C/C++, Python and GAMS models. Additionally, there is an interface implementation at shell script level. Mixed language models as well as distributed models (check Section 5.9) can be run with SimEnv.

<lng>	for model source code
c	C/C++
f	Fortran
py	Python
sh	Shell script level

Tab. 5.2 *Language suffices for SimEnv interface functions (for the GAMS interface check Section 5.7)*


Description Files
Experiment description file <model>.edf
Model output description file <model>.mdf
Model
Model wrapper shell script <model>.run
Fortran model source code (*)
═══════ : associate model description file item with source code item
For transparency reasons: model description item name should be the same as source code item name
model target is original model target value as specified in the previous state-ment to be used as adjusted value if ‘edf_target’ is undefined in <model>.edf
model_target is adjusted model target value
field model_var is output as mdf_var
(*): for C/C++/Python in a likewise manner

```
    ...
    target      edf_target      type                ...
    target      edf_target      default             ...
  { target      edf_target      specific info }
    ...
```

```
    ...
    variable    mdf_var     type                ...
  { variable    mdf_var     coords              ...
    variable    mdf_var     index_extents ... }
    ...
```

```
. $SE_HOME/simenv_ini_sh
...
# perform model with model source code from below:
model
...
. $SE_HOME/simenv_end_sh
```

```
program model
...
integer*4       simenv_ini_f, simenv_get_f, simenv_put_f, simenv_end_f, simenv_sts
real*4          model_target        ! source code item
dimension       model_var ( ... )   ! source code item
...
simenv_sts =    simenv_ini_f ( )
...
model_target =  ...
simenv_sts =    simenv_get_f ( ‘edf_target‘ , model_target , model_target )
...
model_var( ... ) = ...
simenv_sts =    simenv_put_f ( ‘mdf_var‘ , model_var )
...
simenv_sts =    simenv_end_f ( )
...
end
```

Fig. 5.1 *Conceptual scheme of the model interface for C/C++/Fortran/Python*

5.2 Grid and Coordinate 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.

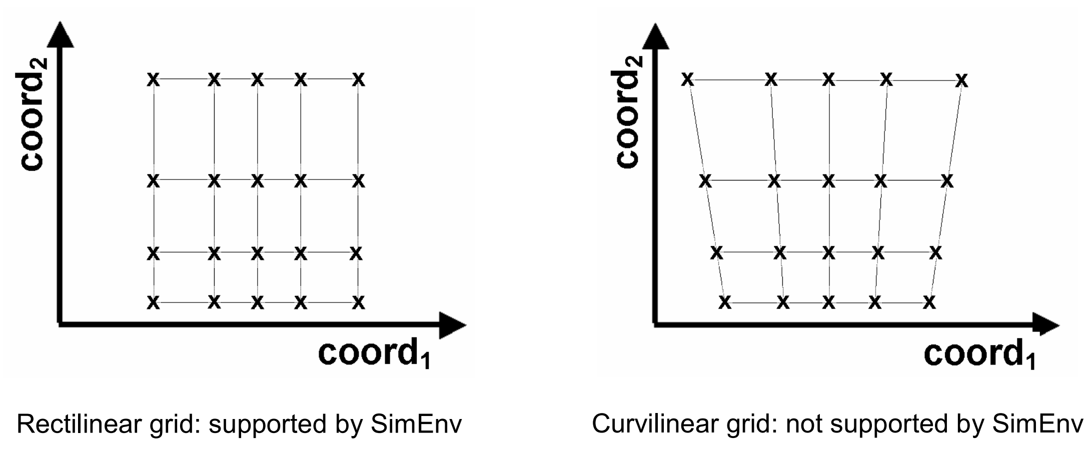

coord$_2$
coord$_1$
Rectilinear grid: supported by SimEnv
coord$_2$
coord$_1$
Curvilinear grid: not supported by SimEnv

Fig. 5.2 *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 experiment 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 output variable values then exist on the grid, spanned up from the coordinate values of the coordinate axes (see Fig. 5.2).

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

5.3 Model Output Description File <model>.mdf

In the model output description file <model>.mdf the model output 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 output variables. Consequently, a model output variable always is defined on a coordinate system, formed from the assigned coordinates to the variable.

keyword	name	sub-keyword	Line type	Max. line nmb.	value	Explanation
general	<nil>	descr	o	any	<string>	model output description
coordinate	<coordinate_ name> (<co_name>)	descr	o	1	<string>	coordinate axis description
		unit	o	1	<string>	coordinate axis unit
		values	m	1	<val_list>	strictly monotonic sequence of coordinate values <co_vals> (for syntax see Tab. 11.6)
variable	<variable_ name>	descr	o	1	<string>	variable description
		unit	o	1	<string>	variable unit
		type	m	1	see Tab. 5.4	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 real values from each coordinate axis to the variable. If missing all coordinate values will be used from all assigned coordinates.
		index_extents	c1	1	<in_val$_{11}$>: <in_val$_{12}$> ,..., <in_val$_{n1}$>: <in_val$_{n2}$>	assigns integer value start and end indeces for each dimension to the variable. Indices can be used to address the variable during experiment post-processing.

Tab. 5.3 *Elements of a model output description file <model>.mdf*

Each model output 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 model interface function simenv_put_* (see below).

<model>.mdf is an ASCII file that holds this information. It follows the coding rules in Section 11.1 on page 119 with the keywords, names, sub-keywords, and values as in Tab. 5.3.

To Tab. 5.3 the following additional rules and explanations apply:

- For the description of **line type** check Tab. 11.4 on page 121.
- Coordinate and variable names must differ from target names in experiment description (see Section 6.1) and from built-in and user-defined operator names for experiment post-processing (see Section 8.5.4).
- Assignment of coordinate axes to variable dimensions and consequently of a grid to a variables is only valid for experiment 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 output 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 values 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 '**index_extents**' portions of variables are made addressable during SimEnv experiment post-processing. In the same way multi-dimensional variables are equipped with indices in the simulation model they also have an index description in the model output description file for purposes of experiment post-processing. It is advisable, that these two descriptions coincide. The index range is described by a start and an end integer value index <in_val$_{i1}$> and/or <in_val_ext$_{i2}$>.
  The index set is a strictly increasing, equidistant set of integer values with an index increment of 1,
  <in_val$_{i1}$> < <in_val$_{i2}$> ,
  <in_val$_{i1}$> ≤ 0 is possible.
- Coordinate values <co_val> and index values <in_val> 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 (synonyms)		Description		Restriction
byte	int*1	1 byte	integer	not for Python models
short	int*2	2 bytes	integer	not for Python models
int	int*4	4 bytes	integer	
float	real*4	4 bytes	real	
double	real*8	8 bytes	real	not for Python models

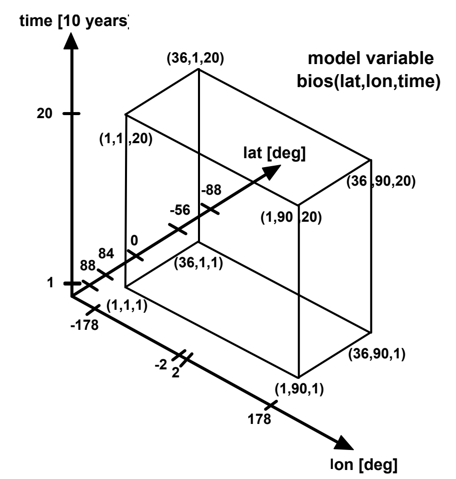
Tab. 5.4 *SimEnv data types*

For the following example of a model output description file and the assigned grids for model output variables check Example 1.1 on page 4:

```
general                descr         World with a resolution of
general                descr         4° lat    x 4° lon x
general                descr         4 levels  x 20 time steps
general                descr         Data centred per lat-lon cell
general                descr         This file is valid for all models
general                descr         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     index_extents 1:45 , 1:90 , 1:4   , 1:20

variable     bios     descr         aggregated biospheric state
variable     bios     unit          g/m^2
variable     bios     type          float
variable     bios     coords        lat      , lon        , time
variable     bios     coord_extents 84.:-56. , -178.:178. , 1:20
variable     bios     index_extents 1:36     , 1:90       , 1:20

variable     atmo_g   type          int
variable     atmo_g   coords        time
variable     atmo_g   index_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 output variable bios refers to Example 5.1 above.
The triples at the edges of the grid are the indices of model output variable bios(lat,lon,time) for the appropriate grid cells.

Fig. 5.3 *Model output variable definition: Grid assignment*

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

Tab. 5.5 describes the model interface functions that can be used in user models written in Fortran or C/C++ (postfix f for Fortran, c for 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. In this table the input and output data types are documented for functions used in Fortran. For C/C++ the corresponding data types are valid.
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.

Function name	Function description	Arguments / function value	Argument / function value description
simenv_ ini_[f | c] ( )	initialize model coupling interface **Perform always as the first SimEnv function in the model. Alternatively include <model>_ [f | c].inc for semi-automated model interface**	integer*4 simenv_ini_ [f | c] (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_ get_[f | c] (target_name, target_def_val, target_adj_val)	get the numerical adjustment in the current single run for the target to be experimented with	character*(*) target_name (input)	name of the target in <model>.edf
		real*4 target_def_val (input)	nominal / default (non-adjusted) target value. If target_name is not defined in <model>.edf then target_adj_val is set to target_def_val
		real*4 target_adj_val (output)	adjusted target value
		integer*4 simenv_get_ [f | c] (function value)	return code = 0 ok = 1 target_name undefined: target_adj_val := target_def_val
simenv_ get_run_[f | c] (run_int, run_char)	get run number of the 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 _[f | c] (function value)	return code = 0 ok
simenv_ put_[f | c] (var_name, field)	output model re-sults 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 simula-tion results
		integer*4 simenv_put_ [f | c] (function value)	return code = 0 ok = 1 var_name undefined = 2 I/O error for model output file

Function name	Function description	Arguments / function value	Argument / function value description
simenv_ slice_[f | c] (var_name, idim, ifrom, ito)	announce to output at the next corresponding simenv_put_[f | c] call only a slice of variable var_name. This announcement becomes inactive after performance of the corresponding simenv_put[f | c]	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. ifrom corresponds to an index from index_extents in <model>.mdf
		integer*4 ito (input)	slice to end at position ito. ito corresponds to an index from index_extents in <model>.mdf
		integer*4 simenv_slice_ [f | c] (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
simenv_ end_[f | c] ( )	close model coupling interface **Perform always the last SimEnv function in the model**	integer*4 simenv_end_ [f | c] (function value)	return code = 0 ok = 2 I/O error for model output file

Tab. 5.5 *Model interface functions for Fortran and C/C++ models*

- Make sure consistency of type and dimension declarations between the model output variables in model source code and the corresponding variable declarations in the model output description file <model>.mdf.
- Model output variables that are not output completely or partially within the user model are handled in experiment 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.
- The include file simenv_mod_[f | c].inc from the SimEnv home directory can be used in a model to declare the SimEnv model interface functions as integer*4 / int for Fortran and/or C/C++. Addionally, these include file declare for the semi-automated model interface (see Section 5.8) auxiliary variables. For the contents of the include files check Tab. 10.5.
- Apply the shell script
  simenv_mod_[f | c | cpp].lnk <model_name>
  From the SimEnv home directory to compile and link an interfaced model
- User models implemented in C/C++ or Fortran have to be linked with the following libraries to interface them to the simulation environment
  - $SE_HOME/libsimenv.a
  - /usr/local/lib/libnetcdf.a
- Tab. 15.12 lists the additionally used symbols when interfacing a Fortran or C/C++ model to SimEnv.
- In
  - Example 15.1 on page 144 the model world_f.f
  - Example 15.3 on page 147 the model world_c.c
  - Example 15.4 on page 149 the model world_cpp.cpp

  are explained.

5.5 Model Interface for Python Models

Function name	Function description	Arguments / function value	Argument / function value description
simenv_ ini_py ( )	initialize model coupling interface **Perform always as the first SimEnv function in the model. Alternatively include <model>_py.inc for semi-automated model interface**	string ini_py (function value)	return code of the spawn function for a SimEnv executable
simenv_ get_py (target_name, target_def_val))	get the numerical adjustment in the current single run for the target to be experimented with	string target_name (input)	name of the target in <model>.edf
		float target_def_val (input)	nominal / default (non-adjusted) target value. If target_name is not defined in <model>.edf then target_adj_val is set to target_def_val
		float get_py (function value)	adjusted target value target_adj_val
simenv_ get_run_py ( )	get the run number of the 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_ put_py (var_name, field)	output 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**		
simenv_ end_py ( )	close model coupling interface **Perform always as the last SimEnv function in the model**		

Tab. 5.6 *Model interface functions for Python models*

Due to the special features of Python the coupling interface to SimEnv differs from that for Fortran and C/C++ in Section 5.4. Additionally, Python supports only some data types (check Tab. 5.4). Tab. 5.6 summarizes the model interface functions for a Python model.

- Python model interface functions are declared in the file simenv.py in the SimEnv home directory. To use these functions in a Python model import it by
  from simenv import *
  and refer to it for example by simenv_get_py.
- Errors that occur during performance of one of the above functions are directly reported to the log-file <model>.nlog.

In Example 15.5 on page 150 the model world_py.py is described in detail.

5.5.1 Standard Shell Scripts for Python Models

<model>.ini
<model>.ini (see Section 7.1 on page 53) is for Python models a mandatory shell 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 with a Python model (check Section 7.3 on page 55) <model>.ini has to be performed again. To force this specify in <model>.cfg (check Section 10.1 on page 107) for the sub-keyword ‘restart_ini’ the value “yes”.

5.6 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 by a set of dot scripts: The shell script <model>.run (see Section 7.1 on page 53) is used to wrap the model and optionally to have at disposal corresponding functionality of the SimEnv model interface functions of Tab. 5.5.

Dot script name	Command description	Arguments	Argument description
$SE_HOME/ simenv_ ini_sh	initialize current single run **Perform always and as the first SimEnv dot script in <model>.run and <model>.rst. Alternatively perform for <model>.run dot script $SE_WS/ <model>_sh.inc for semi-automated model interface**	SE_RUN (output)	operating system environment variable SE_RUN is set to the current run number of the simulation experiment
target_name= '...' target_def_val= ... $SE_HOME/ simenv_ get_sh	get a numerical adjustment in the current single run for the target to be experimented with	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_adj_val is set to target_def_val
		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_adj_val.
$SE_HOME/ simenv_ get_run_sh	get the run number of the 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_ put_sh	**Not available at shell script level**		**Write a model related simenv_put_sh at the language level using the SimEnv model interface functions from Tab. 5.5 or Tab. 5.6**
$SE_HOME/ simenv_ slice_sh	**Not available at shell script level**		
$SE_HOME/ simenv_ end_sh	wrap up current single run **Perform always and as the last SimEnv dot script in <model>.run and <model>.rst**		

Tab. 5.7 *Model interface functions at shell script level*

- 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 the model command line in Unix or Linux. Another way could be to read the targets from a special file in a special file format.
- While for the C/C++/Fortran/Python model interface the names of corresponding targets in the model description file <model>.edf and the model source code can differ and are associated by the first argu-

ment of the interface function simenv_put_* (see Fig. 5.1) the names have to coincide for the model interface at the shell script level.
- Directly before performing the dot script $SE_HOME/simenv_get_sh make sure that the shell script variables target_name and target_def_val have been specified. At the end of the dot script simenv_get_sh these variables are set again to empty strings.
- After running the dot script $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 a model related simenv_put_sh as a transformation program that reads in all the native model output and outputs it to SimEnv by applying the model interface functions simenv_*_* from the SimEnv model interfaces at language level.
- Tab. 10.10 lists the built-in (pre-defined) shell script variables that are defined and/or used by the dot scripts $SE_HOME/simenv_*_sh and are directly available in <model>.run.
- Please notice:
  To perform a dot script (see the Glossary at the end of this document) it has to be preceded by a dot and a space.

In Example 15.6 on page 151 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*

5.7 Model Interface for GAMS Models

SimEnv allows to interface GAMS models to the experiment shell. A GAMS model for SimEnv can consist of a GAMS main model and GAMS sub-models.
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 to SimEnv. 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 workspace. Additional GAMS sub-programs (included files) are not affected bei SimEnv, but one should keep in mind that the GAMS code within SimEnv will be executed in a sub-directory of the current workspace (see below) and so the include statements have to be changed, if the files are addressed in a relative manner (see below).

- The include files are
  <GAMS_model>_simenv_get.inc
  <GAMS_model>_simenv_put.inc
  where <GAMS_model> is the name of a GAMS model file without extension .gms under consideration.
- 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 model interface functions at the language level (see Section 5.4).
- The GAMS include statement $include <GAMS_model>_simenv_get.inc has to be placed in the GAMS model file at such a position where all the GAMS variables are declared. Directly before the include statement the target default values have to be assigned to target variables, that are introduced additionally in the model. Directly after the include statement the target variables with the adjusted target values have to be assigned to the model output variables.
- The GAMS 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.
- 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 workspace. Transformed GAMS models and sub-models are copied to this sub-directory and are performed from there. Keep this in mind when specifying in any GAMS model include statements with relative paths.

In Example 15.8 on page 154 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 sub-models and assigned targets and model output 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 above 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.

5.7.1 Standard Shell Scripts for GAMS Models

<model>.ini
<model>.ini (see Section 7.1 on page 53) is for GAMS models a mandatory shell 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 with a GAMS model (check Section 7.3 on page 55) <model>.ini has to be performed again. To force this specify in <model>.cfg (check Section 10.1 on page 107) for the sub-keyword 'restart_ini' the value "yes".

<model>.run
<model>.run (see Section 7.1 on page 53) has for each GAMS model the same contents:

```
#! /bin/sh
. $SE_HOME/simenv_ini_sh
. $SE_HOME/gams_model_run
. $SE_HOME/simenv_end_sh
```

<model>.end
<model>.end (see Section 7.1 on page 53) is for GAMS models a mandatory shell script and has to have the contents for all GAMS models:

```
$SE_HOME/gams_model_end

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

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

<model>.edf
While for the C/C++/Fortran/Python model interface the names of corresponding targets in the model description file <model>.edf and the model source code can differ and are associated by the first argument of the interface function simenv_put_* (see Fig. 5.1) the names have to coincide for the GAMS model interface. 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 has to be always of type float in the model output description file. In GAMS model code the model output variables declared in the model output description file can be of the numeric types VARIABLES or PARAMETER. The maximum dimensionality of GAMS model output is restricted to 4.

With respect to Example 15.8 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        index_extents 1:2
variable     x         descr         shipment quantities
variable     x         unit          cases
variable     x         type          float
variable     x         coords        plant , market
variable     x         index_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*

5.7.2 GAMS Description File <model>.gdf

The ASCII GAMS description file <model>.gdf is intented to create a block of lines for each GAMS sub-model 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 output 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 Section 11.1 on page 119 with the keywords, names, sub-keywords, and values as in Tab. 5.3.

keyword	name	sub-keyword	Line type	Max. line nmb.	value	Explanation
general	<nil>	descr	o	any	<string>	GAMS coupling description
		keep_runs	o	1	<val_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. 11.6)
		time_limit	o	1	<int_val>	CPU limit in seconds for each GAMS model single run
		options	o	1	<string>	string of options, GAMS main model is started with from command line
model	<model_ name> (without extension .gms)	descr	o	1	<string>	(sub-)model output description
		type	m	1	[main | sub]	identifies GAMS main or sub-model
		get	m	exactly number of targets	<target_name>	get resulting adjustment for <target_name> to this model

keyword	name	sub-keyword	Line type	Max. line nmb.	value	Explanation
		put	m	exactly number of model output 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 interfaced from GAMS to SimEnv ac-cording to <format>

Tab. 5.8 *Elements of a GAMS description file <model>.gdf*

To Tab. 5.8 the following additional rules and explanations apply:

- For the description of **line type** check Tab. 11.4 on page 121.
- Each target and each model output variable as declared in <model>.edf and <model>.mdf respectively has to be used in the value-field of <model>.gdf exactly one time.
- To each GAMS model <model_name> an arbitrary number of targets and model output variables can be assigned to by the corresponding sub-keyword ‘get’ and/or ‘put’.
  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 sub-keyword ‘get’ and ‘put’. This is useful when the main model simply calls sub-models.
- Each model <model_name> in <model>.gdf with at least one sub-keyword ‘get’ 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 sub-keyword ‘put’ 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 sub-keyword ‘type’ value “main”. All other models have to be of sub-keyword type value “sub”.
- The value-field for the sub-keyword ‘put’ is adapted to GAMS syntax to output GAMS model output vari-ables. Afterwards this output is used to generate the appropriate SimEnv output.
  <index_set> is mandatory for variables with a dimensionality > 0. Otherwise, specification of <in-dex_set> is forbidden. Indices as used in the GAMS model are separated from each other by comma.
- The sub-keyword ‘time_limit’ enables limitation of each GAMS model single run in the run ensemble to a maximum CPU-time consumption. If this threshold is reached the single run is aborted and the following single run started. In general, SimEnv nodata values will be assigned to the results of the aborted single runs. The sub-keyword ‘time_limit’ can be necessary since each GAMS model single run itself is an op-timization procedure which could result in an unfeasible CPU time consumption. If the sub-keyword is not used in the gdf-file CPU-time consumption per single run is unlimited.

With respect to Example 15.8 the GAMS description file could look like

```
general                  descr            GAMS model output description
general                  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 GAMS 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 coupled GAMS models*

5.7.3 Files Created during GAMS Model Performance

Additionally to the files listed in Tab. 10.7, during the performance of a GAMS model the files <gams_model>_[pre | main | post].inc are created temporarily in the current workspace 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.

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 workspace. 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 and its performance.

Unlike the other interface implementations GAMS main model terminal output for each single run is redirected to the log-file <model>.nlog in the directory run<run_char>. The modeler is responsible for re-direction of the terminal output from sub-models and from solvers. It is recommended to call all GAMS sub-models with the GAMS option string

```
ll=0 lo=2 lf=<model>.nlog dp=0
```

(see Example 15.8) which is also applied for the main model. With the options sub-keyword 'options' additional options can be specified in <model>.cfg for the main model.

5.8 Semi-Automated Model Interface

Source code manipulations of a model for interfacing it to SimEnv can be classified into four parts:

• Initialization:	simenv_ini_* and simenv_get_run_*
• Target adjustments:	simenv_get_*
• Model output:	simenv_slice_* and simenv_put
• End:	simenv_end_*

Often, initialization and target adjustments can be lumped together in a source code sequence where the target adjustment part has to be updated when new targets are defined in an experiment description file and have to be mapped to model internal targets the first time. Contrarily, model output and end are often distributed in the model source code but do not change so often.

Recognising this situation SimEnv offers beside the standard hand-coded model interface a semi-automated model interface: Initialization and target adjustments are generated automatically during experiment preparation as sequences of source code based on the current experiment description file (and consequently the current experiment targets) for all supported model source code languages but GAMS.

These source code sequences can be used

- for Fortran/C/C++/Python model source codes
  as include files in the model source code and/or
- for the model interface at the shell script level
  as a dot script in <model>.run

to interface the model and consequently to run the experiment with an up-to-date part for initialization and target adjustment. For

- Fortran/C/C++ models
  the model has to be compiled and linked anew with a new include file. This is supported by SimEnv in the course of experiment preparation.
- Python models and the model interface at shell script level
  the include file and/or dot script can be used directly

Generating source code sequences for the semi-automated model interface is invoked by the sub-keyword 'auto_interface' of the keyword 'model' in the model configuration file <model>.cfg (see Section 10.1).

The Fortan/C/C++/Python model interfaces offer to use different names of corresponding targets in the model description file <model>.edf and in the model source code that are associated by the first argument of the interface function simenv_put_* (see Fig. 5.1). **When using the semi-automated model interface the SimEnv target names and the corresponding source code entity names have to be coincided.**

Automatically generated source code sequences are stored in files <model>_[f | c | py | sh].inc in the current workspace $SE_WS. When using targets t1 and t2 in the experiment description file <model>.edf then the source code sequcences have the following contents:

for Fortran:
file <model>_f.inc

```
simenv_sts = simenv_ini_f ( )
simenv_sts = simenv_get_run_f ( simenv_run_int , simenv_run_char )
simenv_sts = simenv_get_f ( 't1' , 0. , t1 )
simenv_sts = simenv_get_f ( 't2' , 0. , t2 )
```

for C/C++:
file <model>_c.inc

```
simenv_sts = simenv_ini_c ( )
simenv_sts = simenv_get_run_c ( &simenv_run_int , simenv_run_char )
simenv_sts = simenv_get_c ( "t1" , &simenv_zero , &t1 )
simenv_sts = simenv_get_c ( "t2" , &simenv_zero , &t2 )
```

for Python:
file <model>_py.inc

```
from simenv import *
simenv_ini_py ( )
simenv_run_int = int ( simenv_get_run_py ( ) )
t1 = float ( simenv_get_py ( ‘t1’ , 0. ) )
t2 = float ( simenv_get_py ( ‘t2’ , 0. ) )
```

for the model interface at shell script level:
file <model>_sh.inc

```
. $SE_HOME/simenv_ini_sh
. $SE_HOME/simenv_get_run_sh
target_name='t1'
target_def_val=0.
. $SE_HOME/simenv_get_sh
target_name='t2'
target_def_val=0.
. $SE_HOME/simenv_get_sh
```

The sequence of targets in the code sequences corresponds to the sequence of targets in the experiment description file <model>.edf. For more than two targets the code sequences are enlarged accordingly.

For the Fortran/C/C++ model interface

- the variables simenv_sts, simenv_run_int, simenv_run_char, and simenv_zero are defined in the model source code include file simenv_mod_[f | c].inc (see Section 10.3).
- model link files <model>.lnk can be declared in the current workspace to link the model anew using the generated code sequences in the course of experiment preparation (for service simenv.run, but not for simenv.rst).

The source code sequences are included in the model source code by

for Fortran:	include ‘<model>_f.inc’
for C/C++:	#include “<model>_c.inc”
for Python:	from simenv import *
for the model interface at shell script level:	. $SE_WS/<model>_sh.inc

Examples can be found in Example 15.2 and Example 15.7.

5.9 Supported Model Structures

SimEnv supports performance of lumped, distributed and parallel models. Information about model structure is specified in the model configuration file <model>.cfg (see Section 10.1) by the sub-keyword ‘structure’. Lumped (standard) models are normally represented by one stand-alone executable. A distributed model in SimEnv consists from a web of stand-alone sub-models, i.e., the model dynamics are computed by performing a set of stand-alone sub-models that normally interact with each other and exchange information. For a parallel model each single run of an experiment needs a set of assign processors.

Lumped (standard) models use in the common sense SimEnv model interface functionality.

For distributed models each of the sub-models can use SimEnv model interface functionality, i.e., simenv_get_*, simenv_get_run_*, simenv_put_*, or simenv_slice_*. In each sub-model with SimEnv model interface functionality simenv_ini_* and simenv_end_* calls have to be incorporated in. Sub-models can be implemented in different programming languages. Additionally, the corresponding SimEnv model interface functionality at shell script level (simenv_*_sh dot scripts) can be applied. As usual, the overall model is wrapped into a shell script <model>.run (see Chapter 7).
The model output description file <model>.mdf collects all the model output variables from all sub-models and the experiment description file <model>.edf collects all the targets from all sub-models.
Announce a distributed model to the simulation environment if

- More than one sub-model uses SimEnv model interface functionality by the simenv_*_*-functions and
- Sub-models get target data from and put model output data to SimEnv data files in parallel. A distributed model where the sub-models are performed sequentially one by one in a cascade-like manner can run in standard mode.

SimEnv interfaced sub-models of a distributed model can reside on different machines. The only prerequisite is that the current workspace and the model output directory can be mapped to each of these machines.

To perform a parallel model within SimEnv simply use the same approach for wrapping a model by the shell script file <model>.run as for standard and distributed models. Instead performing the model within <model>.run submit the model to the load leveler LoadL by using the llsubmit command. Start an experiment from a login-node of the compute cluster and run the experiment at the login machine. SimEnv submits from the login machine all single runs to LoadL and directly finishes afterwards. The parallel operating environment POE and the load leveler LoadL then take responsibility to perform the single model runs.
For the parallel modus the temporary SimEnv files <model>.*_bin and simenv_*.tmp are not deleted at experiment end, i.e. after all single model runs are submitted. These files can be removed manually after finishing the last single run by POE. Check the LoadL services for the end of the last parallel single model run.
To support bookkeeping of SimEnv applications on PIK's parallel cluster machine please insert into the job control file to submit a single model run (file my_parallel_model.jcf in the example below) the line

# @ comment = SimEnv Application

To perform a parallel model in SimEnv the corresponding shell script <model>.run (see Section 7.1 for more information) could have the following contents:

```
#! /bin/sh
. $SE_HOME/simenv_ini_sh

# run a single run of the model:
llsubmit my_parallel_model.jcf

. $SE_HOME/simenv_end_sh
```

Example 5.6 *Shell script <model>.run for a parallel model*

Set the model sub-keyword 'structure' also to "parallel" if the model is to be started in the background (e.g., by my_model &) within <model>.run.

5.10 Using Interfaced Models Outside SimEnv

To run a model interfaced to SimEnv outside the simulation environment in its native mode as before code adaptation the following simple changes 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.
  For this library
  - SimEnv model interface function values (return codes) are 0
  - simenv_get_* forwards target_def_val to target_adj_val
  - simenv_get_run_* returns integer run number 0 and character run string '   ' (six blanks).

- For Python models:
  Replace in the model source code
  ```
  from simenv import *
  ```
  by
  ```
  from simenvdummy import *
  ```
  For this module
  - SimEnv model interface function values (return codes) are 0
  - simenv_get_py forwards target_def_val to target_adj_val
  - simenv_get_run_py returns run 000000.

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

6 Experiment Preparation

Experiment preparation is the first step in experiment performance of a model interfaced 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 General Approach - Experiment Description File <model>.edf

Pre-formed experiment types are the backbone of the SimEnv approach how to use models. They represent in a generic way experiment tasks that have to be equipped with structural in formation from the model and numerical information (see Chapter 4). An equipped experiment type is represented by an experiment description file <model>.edf.

<model>.edf is an ASCII file that follows the coding rules in Section 11.1 on page 119 with the keywords, names, sub-keywords, and value as in Tab. 6.1.

keyword	name	sub-keyword	Line type	Max. line nmb.	value	Explanation
general	<nil>	descr	o	any	<string>	experiment description
		type	m	1	[behaviour | monte carlo | local sensitivity | optimization]	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: specifies how to modify the sampled values by the target default value in simenv_get_*
		default	m	1	<real_val>	target default value <target_def_val>
		adjusts	c3	1	<experiment specific>	specifies how to sample the target to get adjustment values <adj_val>
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. 11.4 on page 121.
- 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 interface function simenv_get_* in the model.
- Target names must differ from model output variables and coordinate names in the model output description file (see Section 5.1) and from built-in and user-defined operator names for experiment post-processing (see Section 8.5.4).

- **To derive the adjusted value the default value as specified in <model>.edf and not the default value from the model code is used in the model interface function simenv_get_*.**
- For adjustment types multiply and relative default <real_val> = 0. is forbidden.
- All experiment specific information is explained in the appropriate sections.
- Specify at least one experiment target.
- When preparing an experiment an experiment input file <model>.edf_adj is generated with the sampled adjustment values <adj_val> according to the information in the sub-keyword 'adjusts'. These values are applied within the interface function simenv_get_* 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 corresponds with the sequence of targets in the target name space (see Section 11.1 on page 119), 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 <run_int> = 0 and <run_char> = 000000 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:
	To derive the final adjusted value <target_adj_val> to use in the model from the sampled adjustment value <adj_val> (from <model>.edf_adj) and the target default value <target_def_val> (as defined in <model>.edf) within the SimEnv model interface function simenv_get_* the sampled adjustment value is …
set	… set to the adjusted target value: **<target_adj_val> = <adj_val>**
add	… added to the target default value: **<target_adj_val> = <adj_val> + <target_def_val>**
multiply	… multiplied by the target default value: **<target_adj_val> = <adj_val> * <target_def_val>**
relative	… increased by 1 and afterwards multiplied by the target default value: **<target_adj_val> = (1. + <adj_val>) * <target_def_val>**

Tab. 6.2 *Adjustment types in experiment preparation*

6.2 Behavioural Analysis

The experiment specific information for experiment description files in Tab. 6.1 on page 43 is defined for behavioural analysis as follows:

keyword	name	sub-keyword	Line type	Max. line nmb.	value	Explanation
target	<target_ name>	adjusts	a	1	<val_list>	value list of target value adjustments <adj_val> (for syntax see Tab. 11.6)
specific	<nil>	comb	m	≥ 1	[default | <combination> | file {<directory>/} <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. 11.4 on page 121.

- For sub-keyword ‘comb’ the following rule holds:

value = [default | <combination>]	for available	sub-keyword ‘adjusts’
value = [file {<directory>/}<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 {<directory>/}<file_name> can be multiple defined for each target.

The sequence of the single runs is determined by the sub-keyword ‘comb’.

6.2.1 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 sub-keywords ‘adjusts’ all over the edf-file.
  - The adjustments are read from the adjustment data file **{<directory>/}<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 Section 11.1 on page 119).
  - Syntax rules for value lists on page 119 hold.
  - Identical adjustments for a target are allowed.
  - During experiment 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.
- To continue a combination <combination> at a following comb-line end the current comb-line by one of the operators “*” or “,”.

6.2.2 Example

The first three experiment description files (a) to (c) represent an experiment description according to Fig. 4.3 (a) to (c) on page 14.

Results in adjusted target values

(a)
```
general          descr      Experiment description for the examples
general          descr      in the SimEnv User Guide (Fig. 4.3 (a))
general          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)
```
general          descr      Fig. 4.3 (b)
general          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)
```
general          descr      Fig. 4.3 (c)
general          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)
```
general          descr      Fig. 4.3 (b)
general          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*

6.3 Monte Carlo Analysis

The experiment specific information for experiment description files in Tab. 6.1 on page 43 is defined for Monte Carlo analysis as follows:

keyword	name	sub-keyword	Line type	Max. line nmb.	value	Explanation
target	<target_ name>	adjusts	m	1	[<distribution> | file {<directory>/} <file_name>]	distribution and distribution parameters to derive a sample of target value adjustments <adj_val> or file name to import an external sample <adj_val>
		sample	c4	1	[random | latin hypercube]	sampling strategy: random or latin hypercube sampling LHS
specific	<nil>	runs	m	1	<int_val>	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. 11.4 on page 121.
- For <distribution> = <distr_shortcut> (<distr_param_1> { , <distr_param_2> }) check Tab. 6.5.
- For implicitly specified distributions according to Tab. 6.5 sample values <adj_val> is generated from the distribution with the assigned distribution parameters.
- For an explicitly ASCII file <file_name> sample values of any distribution are taken directly from this file. For syntax rules for files check Section 11.1. Each record of the ASCII file can hold only one sample value. Sample size has to be identical to <nr_of_runs> from the keyword 'specific'.
- 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 must be greater than 10.

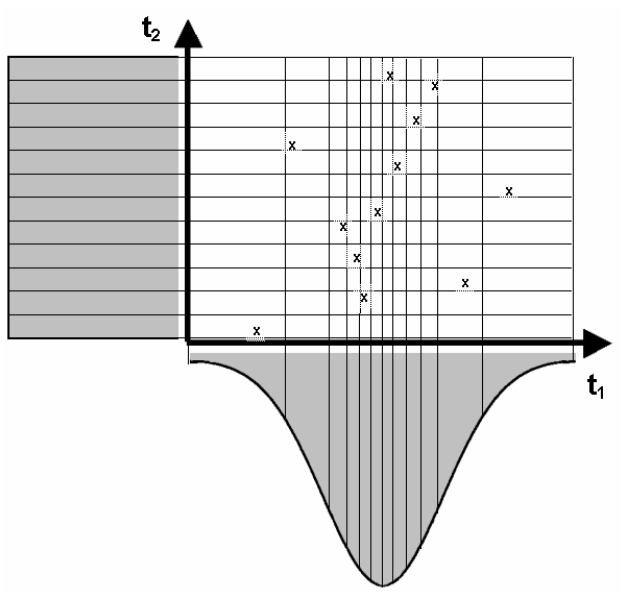
6.3.1 Distribution Functions and their Parameters

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 Section 4.3 and Tab. 4.2.

Be careful when specifying for a Monte Carlo analysis an adjustment type (see Tab. 6.2) that differs from ‘set’. For adjustment type ‘add’ normally the mean value of the sample will be shiftet by the specified target default value <target_def_val>. For adjustment types ‘multiply’ and ‘relative’ the specified distribution will be adulterated normally by the target default value <target_def_val>.



Latin hypercube sampling for a sample size of 12 single simulation runs.

Target t_1 is normally distributed
Target t_2 is uniformly distributed.

Compare with random sampling in Fig. 4.4 on page 14.

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

6.3.2 Example

```
(e) general        descr      Experiment description for the examples
    general        descr      in the SimEnv User Guide
    general        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 sampled from a uniform distrib.
                                                 between 0.5 and 1.5. In simenv_get_*
                                                 each value is multiplied by 2.

    target    p1   type       add
    target    p1   default    1.
    target    p1   sample     random
    target    p1   adjusts    distr N(0,0.4)     p1 is sampled from a normal distribution
                                                 with mean = 1 and variance = 0.4.
                                                 In simenv_get_* each value is increased
                                                 by 1.
```

```
target    p3   type       set
target    p3   default    3.
target    p3   adjusts    file world.dat_e    sample for p3 is read from file
                                              world.dat_e

specific       runs       250
```

Example file: world.edf_e

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

6.4 Local Sensitivity Analysis

The experiment specific information for experiment description files in Tab. 6.1 on page 43 is defined for local sensitivity analysis as follows:

keyword	name	sub-keyword	Line type	Max. line nmb.	value	Explanation
target	<target_ name>	adjusts	f	0		sub-keyword is forbidden for this experiment type
specific	<nil>	incrs	m	1	<val_list>	Increment values that form a sample of target value ad-justments <adj_val> Increments <adj_val> > 0. <adj_val> in <val_list> has to be ordered in a strictly mono-tonic increasing manner. (for syntax see Tab. 11.6)

Tab. 6.6 *Experiment specific elements of an edf-file for local sensitivity analysis*

To Tab. 6.6 the following additional rules and explanations apply:
- For the description of **line type** check Tab. 11.4 on page 121.
- For experiment type local sensitivity analysis only the adjustment types add and relative are allowed.
- Values from the value list must be positive and unique.

6.4.1 Sensitivity Functions and Run Sequence

As an example, the absolute sensitivity function (see Tab. 4.3 on page 17) is as follows:

for adjustment type Add

$$\text{sens\_abs}(\text{<target\_def\_val>},\pm\text{<adj\_val>}) = \frac{z(\text{<target\_def\_val>} \pm \text{<adj\_val>}) - z(\text{<target\_def\_val>})}{\pm \text{<adj\_val>}}$$

for adjustment type Relative

$$\text{sens\_abs}(\text{<target\_def\_val>},\pm\text{<adj\_val>}) = \frac{z(\text{<target\_def\_val>} * (1\pm \text{<adj\_val>}) - z(\text{<target\_def\_val>})}{\pm \text{<target\_def\_val>} * \text{<adj\_val>}}$$

The sequence of the single simulation runs is determined in the following manner:

```
loop        over increment sequence
            loop        over experiment targets
            end loop
end loop
loop        over negative increment sequence
            loop        over experiment targets
            end loop
end loop
```

6.4.2 Example

(f)
```
general          descr      Experiment description for the examples
general          descr      in the SimEnv User Guide
general          type       local sensitivity

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

target     p2    type       relative
target     p2    default    2.
target     p3    type       relative
target     p3    default    3.

specific         incrs      list 0.001,0.01,0.05,0.1
```

Example file: world.edf_f

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

6.5 Optimization

The experiment specific information for experiment description files in Tab. 6.1 on page 43 is defined for local sensitivity analysis as follows:

keyword	name	sub-keyword	Line type	Max. line nmb.	value	Explanation
target	<target_name>	adjusts	m	1	<real_val$_1$>: <real_val$_2$>	lower bound <real_val$_1$> and upper bound <real_val$_2$> to define the target range where the cost function is to be minimized on. <real_val$_1$> ≤ <real_val$_2$> Target adjustment values <adj_val> are sampled in this target range.
specific	<nil>	cost_fct	m	≥ 1	<result>	cost function to minimize. A 0-dimensional result formed according to the rules of the SimEnv post-processor. Do not apply multi-run operators. Cost function definition can be arranged at a series of cost_fct-lines in analogy to the rules for result expressions (see Section 8.1.1).
		max_runs	m	1	<int_val>	number of single runs to end the experiment without checking the other optimization method related stopping criteria.

Tab. 6.7 *Experiment specific elements of an edf-file for an optimization experiment*

To Tab. 6.7 the following additional rules and explanations apply:

- For the description of **line type** check Tab. 11.4 on page 121.

6.5.1 Special Features in Optimization

- This is the only experiment type where the adjustments for the targets of the single runs are not determined before the experiment but in the course of the experiment by the optimization algorithm. Consequently, only the header of the file <model>.edf_adj is created during experiment preparation. The records belonging to the performed single runs are written during experiment performance.
- In parallel to the file model>.edf_adj an ASCII file **<model>.edf_cf** is written during experiment performance with the value of the cost function for each of the single runs.
- The optimization algorithm itself is controlled by additional technical parameters and options that are normally fixed by SimEnv. To modify these settings copy the ASCII file **simenv.opt_opt** from the SimEnv home directory to <model>.opt_opt in the current workspace and edit this file. During the experiment the edited file is used instead of the file with the default constellation in the SimEnv home directory. The description of the options and parameters can be found in Ingber (2004).
- Optimization experiments can not be restarted by the SimEnv service simenv.rst.
- The values for the sub-keywords 'begin_run' and 'end_run' in the configuration file <model>.cfg are ignored for an optimization experiment. The experiment always starts with run number 0 and ends if one of the criteria in the file [<model> **|** simenv].opt_opt (see above) is fulfilled or the explicitly stated end run number from the sub-keyword 'max_runs' <model>.edf is reached.

- As the results of the optimization experiment the optimization return code, the optimal targets, the corresponding value of the cost function and the number of the corresponding single run are documented at the end of the model interface log-file <model>.mlog.
- A protocol from the optimization procedure is delivered by SimEnv in the ASCII file <model>.olog.

6.5.2 Example

```
(g) general          descr      Experiment description for the examples
    general          descr      in the SimEnv User Guide
    general          type       optimization

    target     p1    descr      parameter p1
    target     p1    unit       without
    target     p1    type       set
    target     p1    default    1.
    target     p1    adjusts    -12:12        minimize cost function for p1є <–12 , 12>

    target     p2    type       set
    target     p2    default    2.
    target     p2    adjusts    1:10
    target     p3    type       set
    target     p3    default    3.
    target     p3    adjusts    -12:12
    target     p4    type       set
    target     p4    default    4.
    target     p4    adjusts    1:10

    specific         cost_fct   -sum(bios)    maximize sum(bios) over land masses
    specific         max_runs   700

                                                        Example file: world.edf_g
```

Example 6.4 *Experiment description file <model>.edf for an optimization experiment*

7 Experiment Performance

After experiment preparation experiment performance is the second step in running a model interfaced to SimEnv. Each multi-run experiment can be performed sequentially or in a multi-processor hardware environment. Besides experiment performance from scratch a restart after an experiment interrupt or only for an experiment slice can be handled by SimEnv.

7.1 General Approach

SimEnv enables performance of an experiment on the login-machine or in a job class controlled by the the parallel operating environment POE and the load leveler LoadL. Experiment performance on the login-machine is organized in a way that the single runs of the experiment are performed sequentially. Experiment control by POE and LoadL enables assignment of the simulation load of the single runs of the experiment to a number of processors in distributed, parallel or sequential mode.

Experiments may be performed partially only for a slice out of the run ensemble. Experiment slices are controlled by the general configuration file <model>.cfg by a range of single run numbers.

For successive performance of experiment slices and/or after abnomal experiment interrupt experiments can be re-startet. The experiment log-file <model>.elog is analyzed to identify these single runs out of the run ensemble that have to be performed the first time and/or anew and the corresponding output data structure is appended to the output data that already exists for this experiment.

For all experiment settings the user model has to be wrapped in a shell script <model>.run (see also Fig. 5.1).

- 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 model interface 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 model interface function simenv_put_* (and simenv_slice_*) in the model source code.
- 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 all experiment types a run number 0 with the default values of all experiment targets will is declared additionally to the runs declared in the experiment description file <model>.edf.
- During experiment performance a model interface log-file <model>.mlog is written where the adjustments of experiment target values are stored. All model output to the terminal is re-directed within SimEnv to the experiment model native output log-file <model>.nlog.
- During experiment performance an experiment log-file <model>.elog is written with the minutes of the experiment.
- Do not start / restart / submit another experiment from a workspace where an experiment is still running.
- After the experiment has been finished an e-mail is send on demand (check Section 10.1) to the address as specified in <model>.cfg.
- The status of any running experiment can be acquired by the SimEnv service simenv.sts. For more information check Tab. 10.3.
- For more information check Section 5.1 and Fig. 5.1 and Fig. 7.1.

7.2 Model Wrap Shell Script <model>.run, Experiment-Specific Preparation and Wrap-Up Shell Scripts

- 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 in <model>.run**
    - **#! /bin/sh** **is the first line**
    - **. $SE_HOME/simenv_ini_sh** **is performed always and as the first SimEnv dot script**
    - **. $SE_HOME/simenv_end_sh** **is performed always and as the last SimEnv dot script**

    (see Tab. 5.7 on page 32 and Example 7.1 below).
  - To cancel the whole experiment after the performance of the current single run due to any condition of this run make sure a file **$SE_WS/<model>_$run_char.err** exists as an indicator to stop. 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. 5.7 on page 32 and Example 7.1 below).
    - Touch the file $SE_WS/<model>_$run_char.err.
  - Terminal output from <model>.run is redirected to the log-file <model>.nlog.
  - For GAMS models <model>.run has a pre-defined structure. Check Section 5.7.1 for more information.

- The user can define an optional model specific experiment preparation shell script **<model>.ini** that is performed additionally after standard experiment preparation and before setting up a new experiment. For experiment restart <model>.ini is performed only on request (see Section 7.3 below).
  In <model>.ini additional settings / checks can be performed. For return codes unless 0 from <model>.ini the experiment will not be started.
  Terminal output from <model>.ini is also re-directed to the log-file <model>.nlog.
  For Python and GAMS models <model>.ini is a mandatory shell script with standardized contents. Check Sections 5.5.1 and 5.7.1 for more information.

- After the experiment has been finished the native model specific output from the experiment can be wrapped up with the optional model specific shell script **<model>.end**.
  Terminal output from <model>.end is re-directed to the log-file <model>.nlog.
  For GAMS models <model>.end. is a mandatory shell script with standardized contents. Check Section 5.7.1 for more information.

- All of these three shell scripts have to have execute permission. Ensure this by the Unix / Linux command

  chmod u+x <model>.[run | ini | end]

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

```
#! /bin/sh

# perform always and as the first $SE_HOME/simenv_*_sh dot script:
. $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 workspace $SE_WS
# as an indicator to SimEnv for this.
```

```
        if test $? -ne 0
        then
              . $SE_HOME/simenv_get_run_sh
              touch $SE_WS/world_f_$run_char.err
        fi

        # perform always and as the last $SE_HOME/simenv_*_sh dot script:
        . $SE_HOME/simenv_end_sh
```

Example file: world_f.run

Example 7.1 *Shell script <model>.run to wrap the user model*

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

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*

7.3 Experiment Parallelization

According to the general SimEnv approach how to use a model the single runs of an experiment are independent from each other. The only exception is the experiment type optimization where the adjustments for the current single run are determined on the outcomes of previous single runs. Keeping this in mind, SimEnv enables the parallelization of the experiment in the sense that several single runs can be performed in parallel without influencing each other. This opens an approach for a computer network or a compute cluster of connected machines

- to distribute the single runs of an experiment acroos the network / on the cluster
- to perform the single runs there and
- to collect after the end of the single model run the model output data and related information

SimEnv supports distribution of single runs from an experiment for IBM's cluster architectures. These clusters are managed by the parallel operating environment POE and the load leveler LoadL. The processors of a compute cluster are assigned to job classes where jobs can be submitted to.

Three different distribution stategies are offered by the simulation environment:

Perform the single runs of an experiment …

- … on all the processors of a job class (dis - distributed strategy)
  The single runs are submitted to the job class as single jobs in a way that all available processors of the class can be used. Due to controlling the submit process dynamically, the job class will not be overloaded by the single run jobs of the experiment but the submit process will wait on demand. The submit process itself is started in the background.
  The experiment performance will start when a processor of this job class is free.
  Use this strategy for best utilization of all job class processors.
- … on pre-allocated processors of a class (par - parallel strategy)
  A number of processors are assigned to the experiment during experiment preparation and one parallel job is submitted to the job class. During the experiment one communication processor is responsible for experiment management while the other processors serve as simulation processors for the single runs.
  The experiment performance will start when the assigned number of processors are free in this class.
  Use this strategy to make sure to run an experiment in a certain time.
- … sequentially on one processor of a class (seq - sequential strategy)
  One processor of the job class is assigned to the whole experiment and the experiment is performed single run by single run on this processor as a SimEnv experiment performance on the login machine.
  The experiment performance will start when one processor of this job class is free.
  Use this strategy when the other two strategies are impossible (e.g., for an optimization experiment) and you want to use the hardware sessources of the compute cluster.

For an experiment performance controlled by the parallel operating environment POE and the load leveler LoadL make sure that the environment variables SE_HOME is set in the file $HOME/.profile correctly.

After the experiment is submitted to the load leveler the current login session can be closed.

Default job control files are supplied by SimEnv to ensure communication with POE and the load leveler. These job control files may be copied to the current workspace, 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 the ASCII job control files **simenv.jcf_[dis | par | seq**] from the SimEnv home directory to <model>.jcf_[dis | par | seq] in the current workspace, modify the file(s) according to the needs of the experiment one want to perform and / or the machine one want to use and start afterwards simenv.run and/or simenv.rst. If available in the current workspace, these modified job control files are used instead of the original files in the SimEnv home directory.
[<model> | simenv].jcf_dis submits a job in distributed mode, [<model> | simenv].jcf_seq in sequential mode, and [<model> | simenv].jcf_par in parallel mode.

Default job control files enable automatic restart of the experiment by the load leveler after an interrupt of the job caused by POE, the load leveller, or the operating system. The user does not need to restart the experiment manually after such an event.

For parallel models itself see Section 5.9.

7.4 Experiment Restart

When an experiment was interrupted / has failed due to any reason or in the case of partial experiment performance (see Section 7.5 below) it can be restarted:

- 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 values for the sub-keywords of the keyword 'experiment' 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 by the interrupted / previous new-start 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 and 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 the latter 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 in <model>.rst

• **#! /bin/sh**	**is the first line**
• **. $SE_HOME/simenv_ini_sh**	**is performed always and as the first SimEnv dot script**
• **. $SE_HOME/simenv_end_sh**	**is performed always and as the last SimEnv dot script**

  (see Tab. 5.7 on page 32 and Example 7.4 below).
  Make sure that <model>.rst has execute permission by the Unix / Linux command
  chmod u+x <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 Tab. 10.10).
  Terminal output from <model>.rst is re-directed to the log-file <model>.nlog.
- Experiment restart works without standard SimEnv experiment preparation. Instead, experiment preparation files and other information from the interrupted experiment will be used.
- For a restart, 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’ and its value “yes”.
  For Python and GAMS models interfaced to SimEnv <model>.ini has to be performed mandatorily. Consequently, the value of restart_ini has to be set to “yes” (check Sections 5.5.1 and 5.7.1)
- **<model>.cfg** will be checked anew for experiment restart. Do not change for a restart any of the information related to the keyword ‘model’ in <model>.cfg.
- Minutes of the restarted experiment will be appended to the log-files <model>.mlog, <model>.nlog, and <model>.elog, respectively from the interrupted experiment.
- Restart can be applied to an experiment several times successively.
- Experiment restart can be performed also as an partial experiment, independently on the partial status of the original model
- Experiment re-start is not possible for the experiment type optimization.

For the model world_sh (check Example 15.6 on page 151) the following contents could be defined for the restart shell script world_sh.rst:

```
#! /bin/sh

# perform always and as the first $SE_HOME/simenv_*_sh dot script:
. $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$run_char
then
   rm -fR run$run_char
fi

# perform always and as the last $SE_HOME/simenv_*_sh dot script:
. $SE_HOME/simenv_end_sh
```

Example file: world_sh.rst

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

7.5 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 sub-keywords ‘begin_run’ and ‘end_run’ 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.
- Experiment partial performance is not possible for the experiment type optimization.

7.6 Experiment Related User Shell Scripts and Files

Shell script / file	Explanation	Used for (*)	Exist status
Shell scripts (terminal output is re-directed to <model>.nlog) (**)			
<model>.run	model shell script to wrap the model executable Model interface dot scripts at shell script level simenv_*_sh can / have to be applied in <model>.run: • $SE_HOME/simenv_ini_sh has to be performed always and as the first SimEnv dot script simenv_*_sh • $SE_HOME/simenv_end_sh has to be performed always and as the last SimEnv dot script simenv_*_sh • Pre-defined contents for GAMS models	S R	mandatory
<model>.rst	model shell script to prepare single model run restart for such single runs that were started but not finished during the previous experiment start / restart • $SE_HOME/simenv_ini_sh has to be performed always and as the first SimEnv dot script simenv_*_sh • $SE_HOME/simenv_end_sh has to be performed always and as the last SimEnv dot script simenv_*_sh • $SE_HOME/simenv_get_run_sh can be used	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 shell script is unequal 0 • For experiment re-start <model>.ini will be performed only on request • Pre-defined contents for Python and Gams models	S (R)	optional, for Python and GAMS models mandatory
<model>.end	model shell script to clean up simulation experiment from non-SimEnv files • Pre-defined contents for GAMS models	S R	optional
Files			
<model>_ <run_char>.err	touch such a file during performing the model, in <model>.run and/or in <model>.rst as an indicator to stop the complete experiment after single run <run_char> has been finished	A	optional
<model>.jcf_ [dis | par | seq]	model-specific job control file to submit an experiment in distributed, parallel and/or sequential mode by the load leveler LoadL • Copy from general file $SE_HOME/simenv.jcf_[par | seq] on demand	L	optional
<model>.opt_opt	model-specific control and option file for experiment type Optimization • Copy from general file $SE_HOME/simenv.opt_opt on demand	O	optional

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

(*): shell script applied for
*R: **R**estart of an experiment by simenv.rst <model>*
*S: **S**tart of an experiment by simenv.run <model>*
file applied for
*A: **A**ll experiment perform. at the login machine or by load leveler submission*
*L: **L**oad leveler experiment submission*
*O: **O**ptimization experiment performance*
(**): make sure by the Unix / Linux command chmod u+x <model>.<ext> that the shell script <model>.<ext> has execute permission

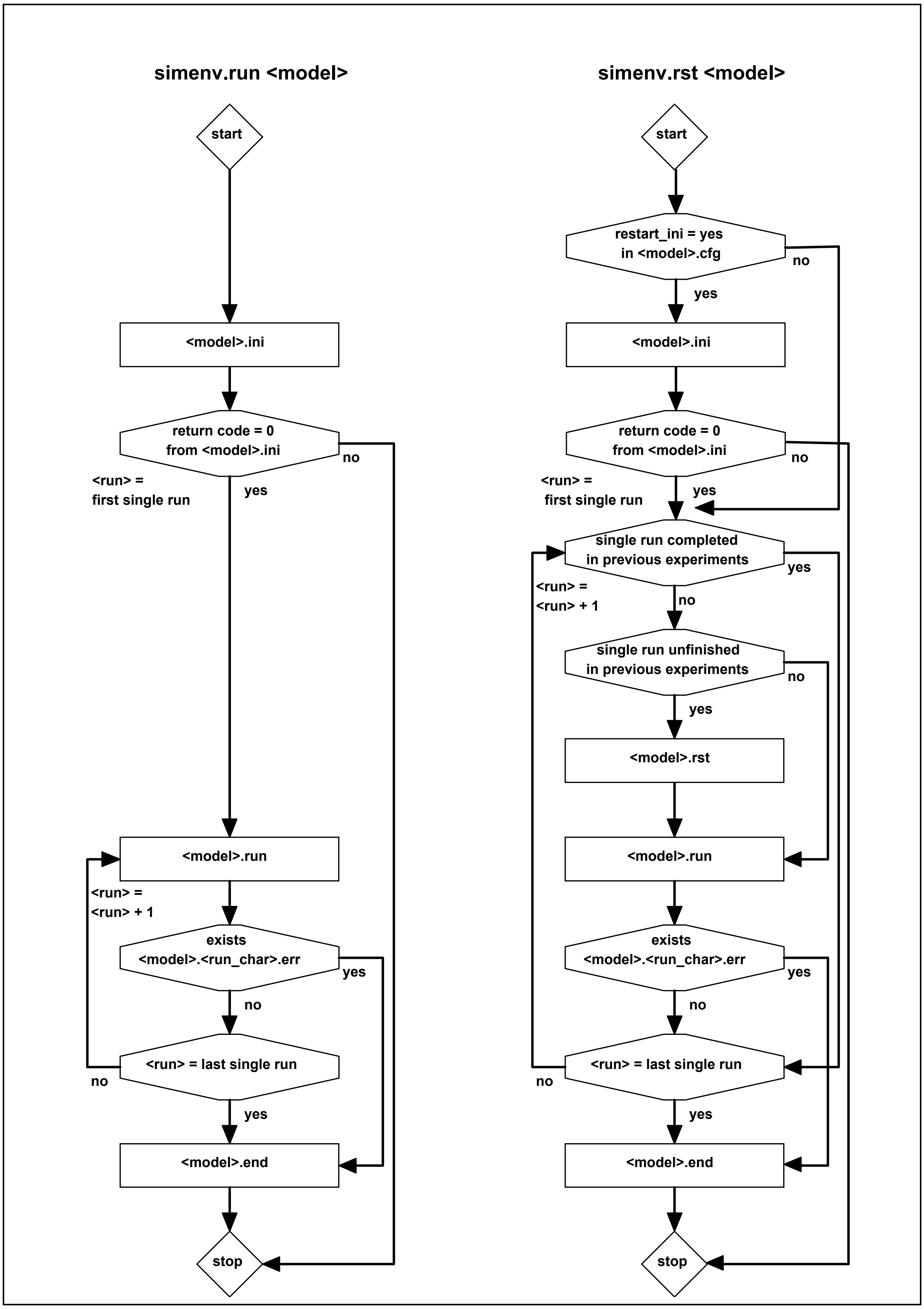


Fig. 7.1 *Flowcharts for performing simenv.run and simenv.rst*
First and last single run always refer to the corresponding settings in <model>.cfg

7.7 Saving Experiments

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

• <mdel>.out[all | <run_char>].[nc | ieee]		from the model output directory
• <model>.cfg		from the current workspace
• <model>.mdf		from the current workspace
• <model>.edf		from the current workspace
• <model>.edf_adj	(for optimization)	from the current workspace
• <model>.edf_cf	(for optimization)	from the current workspace
• <model>.elog	(optional)	from the current workspace
• <model>.mlog	(optional)	from the current workspace
• <model>.nlog	(optional)	from the current workspace
• <model>.jcf_ [dis | par | seq]	(optional)	from the current workspace
• <model>.olog	(optional, for optimization)	from the current workspace
• <model>.opt_opt	(optional, for optimization)	from the current workspace

Do not modify after the experiment in

• <model>.cfg	the information assigned to the keyword ‘model’
• <model>.[mdf | edf]	all information including the sequence of the model output variables and/or experiment targets

8 Experiment Post-Processing

Goal of experiment 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. Additionally, composed operators can be derived from both built-in and user-defined operators. Operator chains and recursions are possible. Macros can be defined as abbreviations for operator chains.

8.1 General Approach

8.1.1 Post-Processor Results

In SimEnv experiment post-processing post-processor results (synonym: output functions) are derived from model output of the experiment and from reference data. A post-processor result is specified by a post-processor expression, optionally prefixed by a result description and a result unit string:

<result> = { { <result_description> } { [<result_unit>] } := } <result_expression>

<result>	by the string “Enter a result” the user is asked to enter a result. Input lines with a character # as the first non-white space character are treated as comments. The experiment post-processing session is finished by entering <ret> or a sequence of white spaces instead of a result. For case sensitivity of <result> check Tab. 10.11 on page 117.
<result_description>	must not contain an apostrophe character “‘”.
<result_unit>	characters “[” and “]” belong to the syntax and are **not** part of the this document convention as defined in Tab. 1.1 Result description and/or unit together with the separator “:=” have to be specified in the first input line. The result expression itself may follow at the following input line.
<result_expression>	is a chain of SimEnv operators applied to model output variables and/or reference data. Can be continued on a new input line (continue expression:) if the current input line ends on one of the operators “+” , “-” , “*” , “/”, or “**” or on the operand separator “,” in operators. White spaces are filtered out from the result expression string, also from character arguments.

<result_description> or <result_unit> are used to describe the result in the corresponding result output file (see Chapter 12). For the case one of these entities is not specified SimEnv analyses the result expression: For a result expression formed without any operator or only from one operator and using exactly one model output variable and/or one experiment target <result_description> and/or <result_unit> is copied from the corresponding information for the sub-keyword ‘descr’ in <model>.mdf (for a model output variable as an operand of this operator) and/or from <model>.edf (for an experiment target as an operand of this operator). The only operator used in this expression must not transform the contents of the operand in general (must be invariant with respect to description and unit). For all other cases <result_description> is set to the string res_<xy> and <result_unit> is undefined.

Having a model output variable definition as in Example 5.1 on page 27 then in experiment post-processing

`abs(atmo)+3`	applies operator abs to atmo and adds 3 (multi-operator result expression) <result_description> = ‘res_<xy>’ <result_unit> undefined
`Energy [MWh] := abs(atmo)+3`	as above, but: <result_description> = ‘Energy’ <result_unit> = ‘MWh’

`Energy [MWh] :=`
`abs(atmo)+`
`3`
	as above

`[MWh] := abs(atmo)+3`	as above, but: <result_description> = ‘res_<xy>’ <result_unit> = ‘MWh’
`sign(atmo)`	applies operator sign to atmo (operator sign is not invariant w.r.t. the contents of its operand) <result_description> = ‘res<xy>’ <result_unit> undefined
`abs(atmo)`	applies operator abs to atmo (operator abs is invariant w.r.t. the contents of its operand) <result_description> = ‘aggregated atmospheric state’ (according to <model>.mdf) <result_unit> = ‘without’ (according to <model>.mdf)
`Energy := abs(atmo)`	applies operator abs to atmo <result_description> = ‘Energy’ (according to <model>.mdf) <result_unit> = ‘without’ (according to <model>.mdf)

Example 8.1 *Addressing results in experiment post-processing*

8.1.2 Operands

Operands in result expressions can be

- Model output variables as defined in <model>.mdf
  In the following abbreviated by **arg**
  Example: `atmo`
- Experiment targets as defined in <model>.edf
  In the following abbreviated by **arg**
  Example: `p1`
- Constants <int_val> or <real_val>
  In the following abbreviated by **int_arg and/or real_arg**
  Example: `12` and `-12` and `12.34` and `-1.234e+1`
- Character strings <string>, enclosed in single quotation marks
  In the following abbreviated by **char_arg**
  Example: `‘tie_avg’`
- Operator results
  In the following abbreviated by **arg**
  Example: `abs(atmo)` and `atmo+3`.
- Macros as defined in <model>.mac (see Section 8.7)
  Example: `equ_100yrs_m`
- Wildcard operands (see Section 8.8)

Example: `&v&`

As for model output variables (see Section 5.1) also to each operand (with the exception of character string operands)

Dimensionality	**dim(operand)**	and
Extents	**ext(operand,i)**	with i=1 ,..., dim(operand)
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 Section 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 Section 8.1.4).
- Consequently, the output of an operator and finally a post-processor result as a sequence of operators applied to operands also has unique dimensionality, extents and coordinates.
- Experiment targets and constants always have a dimensionality of 0.
- Operands of dimensionality 0 and character string operands do not have a coordinate assignment.

8.1.3 Model Output Variables

- A variable of dimensionality n corresponds with a n-dimensional array and is defined at an 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 experiment post-processing (see below). Dimensionality, dimensions and coordinate description of this data field is derived from the model output variable description in <model>.mdf.
- Model output variables are specified in the ASCII model output description file <model>.mdf (see Tab. 5.3 on page 25) by their
  - Name
  - Dimensionality
  - Extents
  - Coordinate assignment to each dimension
  - Data type (see Tab. 5.4 on page 26).
  - Use the service 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 experiment 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 i** 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 a **coordinate range c** 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 Section 10.1 on page 107) a global default for index and/or coordinate addressing is established for the whole experiment post-processing session. This global default can be overwritten locally by using c= and/or i=.

Having a model output variable definition as in Example 5.1 on page 27 then in experiment post-processing result expressions can be

`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 8.2 *Addressing model output variables in experiment post-processing*

8.1.4 Operators

- Operators transform dimensionality, dimensions, and coordinates of the their non-character operator arguments into unique dimensionality, dimensions and coordinates of the operator result (check Section 8.1.2).
  There are
  - Single-argument operators that replicate dimensionality, dimensions and coordinates from the only argument to the operator result
    Example: `sin(atmo)`
  - Multi-argument operators that demand a certain relation between dimensionalities, dimensions and coordinates of their arguments
    Example: `mod(atmo(c=84:-56,*,c=1,*),bios)`
  - Operators that increase the dimensionality of the operator result and assign new coordinates to the additional dimensions (check Tab. 10.9) or form new coordinates from resulting target adjustments
    Example: `ens(atmo)`
- SimEnv experiment post-processing operators may have two special types of arguments:
  - Character arguments char_arg:
    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). Some built-in operators allow an empty string (e.g., behav)
    Example: `count(‘undef’,atmo)`
    `behav(‘ ’,atmo)`
  - Integer or real (float) constant arguments int_arg or float_arg:
    Only constants in appropriate format are valid as arguments. Model output variables of dimensionality 0 or general operands with dimensionality 0 are invalid.
    Example: `move_avg(‘0001’,‘lin’,3,atmo)`
    `qnt(33.333,atmo)`
  - If character and integer/real constant arguments are defined for an operator then there is always the following sequence of the operator arguments:
    { char_arg } { int_arg } { real_arg } { arg }
    Example: `hgr_l(‘1000’,’bin_mid’,20,0.,0.,atmo)`
- Operators are generic with respect to the data types of their operands: Each non-character and non-constant argument can be used with operands of all defined data types (see Section 5.1). Internally, arguments of any type are converted to a float representation. This may lead to undefined arguments of type double in float representation.
- Results of SimEnv experiment post-processing operators are always of the type 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 from each other 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`

8.1.5 Operator Classification, Flexible Coordinate Checking

Tab. 8.1 lists for all built-in operators a classification of argument restrictions and result description that are used in the following for the explanation of built-in operators.

Argument restriction(s) / result description		**Argument restriction(s)**	**Result description (check Section 8.1.2 for syntax)**
(1)		dimensionality, extents and coordinates of the only non-character / non-constant argument arg can be arbitrary	same dimensionality, extents and coordinates as the only non-character / non-constant 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)	all non-character / non-constant arguments arg with same dimensionality, extents and coordinates **(*)**	same dimensionality, extents and coordinates as all the non-character / non-constant 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 / non-constant arguments arg with same non-zero dimensionality, extents and coordinates **(*)**, all the other non-character arguments with dimensionality 0	same dimensionality, extents and coordinates as all the non-character / non-constant 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 0-dimensional argument is applied to each element of the non-zero dimensional argument
(3)		dimensionality, extents and coordinates of the only non-character / non-constant argument can be arbitrary	dim(res) = 0
(4)	(4.1)	all non-character / non-constant arguments with same dimensionality, extents and coordinates **(*)**	dim(res) = 0
	(4.2)	some non-character / non-constant arguments with same non-zero dimensionality, extents and coordinates **(*)**, all the other non-character / non-constant arguments with dimensionality 0	dim(res) = 0 the 0-dimensional argument is applied to each element of the non-zero dimensional argument
(5)		dimensionality, extents and coordinates of the first non-character / non-constant argument arg can be arbitrary, all the other following 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 / non-constant 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	dim(res) = 0

Tab. 8.1 *Classified argument restriction(s) / result description*
(*): for the different levels of checking a coordinate description see below

The requirement for a lot of operators to have same coordinates for same dimensions may restrict application of experiment post-processing especially for hypothesis checking 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 the SimEnv complete post-processing session:

- 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 the appropriate operator result is delivered from its first non-character / non-constant 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 the appropriate operator result is delivered from its first non-character / non-constant operand.

Check Example 8.3 for examples.

Having a model output variable definition as in Example 5.1 on page 27 then the checking rules for coordinates are applied in the following manner to operands with dimensionality 1:

Result 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('bin_no',20,0.,0.,atmo)` (differing coordinate names, same coordinate values)	no	yes	yes
`atmo_g(c=6:16) + atmo_g(c=8:18)` (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 determined by the first summand for coord_check = [weak | without].

Example 8.3 *Checking rules for coordinates*

8.2 Built-In Generic Standard Aggregation / Moment Operators

The generic operators in Tab. 8.2 can be applied during experiment post-processing to derive aggregations and moments from operands in different ways by appending suffixes (_n, _l, _e, without suffix) to the generic operator name or by incorporating them into the filter argument for experiment specific operators of bahavioural analysis:

Generic aggregation and moment operator	**Meaning**
max	maximum of values
min	minimum of values
sum	sum of values
avg	arithmetic 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
maxprop	maximal, suffix related property of values
minprop	minimal, suffix related property of values

Tab. 8.2 *Built-in generic standard aggregation / moment operators*

For more information check Sections 8.3.3 and 8.4.1.

8.3 Built-In Elemental, Basic, and Advanced Operators

8.3.1 Elemental Operators

Name	**Meaning**	**Argument restriction(s) / result description (see Tab. 8.1)**	**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. 8.3 *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.
- For examples check Section 8.3.5.

8.3.2 Basic and Trigonometric Operators

Name	Meaning	Argument restriction(s) / result description (see Tab. 8.1)	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)	exponential function	(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.`
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.`

Tab. 8.4 *Built-in basic and trigonometric operators*

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

- **All operators** 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.
- For examples check Section 8.3.5.

8.3.3 Standard Aggregation / Moment Operators

The generic standard aggregation / moment operators in Tab. 8.2 can be applied during experiment 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 / non-constant argument
  Result is a scalar (an operator result of dimensionality 0) 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 non-character / non-constant arguments with argument restriction(s) / result description according to (2) in Tab. 8.1 on page 68 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 / non-constant argument 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.
- For **examples** check Section 8.3.5.

Aggregation and moment operator	Argument restriction(s) / result description (see Tab. 8.1)
max(arg)	(3)
min(arg)	
sum(arg)	
avg(arg)	
var(arg)	
avgg(arg)	
avgh(arg)	
avgw(arg1,arg2)	(4.1) arg2 = weight
hgr(char_arg1,int_arg2, real_arg3,real_arg4, arg5)	dim(res) = 1 ext(res,dim(res)) = number of bins for char_arg1 = ‘bin_no’ (bin number): coord(res,dim(res)) = name = bin_no values = equidist_end 1(1) number of bins for char_arg1 = ‘bin_mid’ (bin mid): coord(res,dim(res)) = name = bin_mid values = equidist_end 1st bin mid (bin width) number of bins char_arg1 see above int_arg2 = number of bins: 4 ≤ int_arg2 ≤ number_of_values or = 0: automatic determination: number of bins = max(4,number_of_values_of_arg5/10) real_arg3 left bin bound for bin number 1 real_arg4 right bin bound for bin number int_arg2 real_arg3 = real_arg4 = 0.: determine bounds by min(arg5) and max(arg5) min(arg5) = max(arg5): all result values are undefined
count(char_arg1,arg2)	(3) char_arg1 = [all | def | undef]
maxprop(arg)	dim(res) = 1 for dim(arg) > 1 ext(res,1) = dim(arg) dim(res) = 0 else return 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 storage model (see Section 15.7 - Glossary).
minprop(arg)	

Tab. 8.5 *Built-in standard aggregation / moment operators without suffix*

Aggregation and moment operator	Argument restriction(s) / result description (see Tab. 8.1)
max_n(arg1 ,..., argn)	(4)
min_n(arg1 ,..., argn)	
maxprop_n(arg1 ,..., argn)	(4) return per result element the argument position (1 ,..., n) where the extreme is reached the first time. Processing sequence starts with arg1.
minprop_n(arg1 ,..., argn)	

Tab. 8.6 *Built-in standard aggregation / moment operators with suffix _n*

Aggregation and moment operator	Argument restriction(s) / result description	
min_l(char_arg1,arg2)	dim(argi) > 1 ext(argi) = arbitrary dim(res), ext(res,i) according to char_arg1 and argi	
max_l(char_arg1,arg2)		
sum_l(char_arg1,arg2)		
avg_l(char_arg1,arg2)		
var_l(char_arg1,arg2)		
avgg_l(char_arg1,arg2)		
avgh_l(char_arg1,arg2)		

avgw_l(char_arg1,arg2, arg3)		dim(arg2) = dim(arg3)
		ext(arg2,i) = ext(arg3,i)
		arg3 = weight

hgr_l(char_arg1, char_arg2,int_arg3, real_arg4,real_arg5, arg6)		dim(res) = 1 + dim(res) of all other operators ext(res,dim(res)) = number of bins for char_arg2 = 'bin_no' (bin number): coord(res,dim(res)) = name = bin_no values = equidist_end 1(1) number of bins for char_arg2 = 'bin_mid' (bin mid): coord(res,dim(res)) = name = bin_mid values = equidist_end 1st bin mid (bin width) number of bins char_arg2 see above int_arg3 number of bins 4 ≤ int_arg3 ≤ number_of_ values_of_arg6 or 0: automatic determination = max(4,number_of_values/10) real_arg4 left bin bound for bin number 1 real_arg5 right bin bound for bin number int_arg3 real_arg4 = real_arg5 = 0.: de-termine bounds by min(arg6) and max(arg6) min(arg6) = max(arg6): all result values are undefined
count_l(char_arg1, char_arg2,arg3)		char_arg2 = [all | def | undef]
minprop_l(char_arg1, arg2)	as above, but: dim(res) is increased by 1 w.r.t. above. ext(res,dim(res)) = dim(arg2) coord(res,dim(res)): name = index values = equidist_end 1(1)"n"	return the indices of those elements of arg2 where the extreme is reached the first time according to char_arg1 and to a Fortran-like processing sequence / storage model (see Section 15.7 - Glossary) of the argument field arg2.
maxprop_l(char_arg1, arg2)		

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

The loop character argument char_arg1 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

8.3.4 Advanced Operators

Name	Meaning	Argument restriction(s) / result description (see Tab. 8.1)	Argument value restriction	Example

classify(int_arg1, real_arg2, real_arg3,arg4)	classify arg4 into int_arg1 classes;
	potentially restrict classi-fication to interval (real_arg2 , real_arg3).
		(1)
		dim(arg4) > 0
		int_arg1 = number of classes
		2 ≤ int_arg1 ≤ number of values of arg4
		= 0: automatic determina tion:
		number of classes = max(2,number ofvalues/10)
		real_arg2 = minimum bound for values in class # 1
		real_arg3 = maximum bound for values in class # int_arg1
		arg2 = 0. and arg3 = 0.: automatic bound determination
				`classify((10,0.,0.,atmo)`

clip(char_arg1, arg2)	clip arg2 according to char_arg1	dim(arg2) > 0 dim(res), ext(res,i) depend on char_arg1 and arg2 char_arg1 = clip range		`clip('0,*,1,10', atmo)`
cumul(char_arg1, arg2)	cumulate arg2 according to char_arg1	(1) dim(arg2) > 0 char_arg1 = cumulation indicator per dimension		`cumul('0001', atmo)`
flip(char_arg1, arg2)	flip arg2 according to char_arg1	(1), but coordinates are also flipped dim(arg2) > 0 char_arg1 = flip indicator per dimension		`flip('0001', atmo)`

get_data(char_arg1, char_arg2, char_arg3, arg4)	get data from an external file	dimensionality, extents and coordi-nates according to char_arg3 and char_arg4
		char_arg1 = data file format
		= ascii
		char_arg2 = data file name
		char_arg3 = coordinate specification / transformation file name
		char_arg4 = variable to get from the data file
				`get_data('ascii', 'data.asc', 'data.def', variable)`

get_experiment(char_arg1, char_arg2, char_arg3, arg4)	include an other experiment	(1) but coordinates according to char_arg3 char_arg1 = experiment directory char_arg2 = model experimented with char_arg3 = file how to transform result coordinates arg4 = result from the other experiment		`get_experiment('mod_res', 'mod', 'mod.trf', avg(atmo)-400.)`
get_table_fct(char_arg1, arg2)	apply table function with linear interpolation of table char_arg1 to arg2	(1) char_arg1 = file name		`get_table_fct('table.usr', atmo)`

Name	Meaning	Argument restriction(s) / result description (see Tab. 8.1)	Argument value restriction	Example

if(char_arg1,
arg2,arg3,arg4)
	conditional if-construct	(5)
		char_arg1 = comparison operator
		arg2 = comparator
		arg3, arg4 = new assignments
				`if('<',atmo,400,`
				`atmo)`

mask(char_arg1,
arg2,arg3)
	mask values of arg2 (set them undefined) by comparing arg2 and arg3 using operator char_arg1	(5)
		char_arg1 = comparison operator
				`mask('<',atmo,`
				`400)`

matmul(arg1,
arg2)
	matrix multiplication	dim(arg1) = dim(arg2) = dim(res)
		= 2
		ext(res,i) according to matrix multiplication rules
				`matmul(`
				`atmo(*,*,1,1),`
				`transpose('21',`
				`atmo(*,*,1,1)))`

move_avg(
char_arg1,
char_arg2,
int_arg3,arg4)
	moving average of arg4	(1)
		dim(arg4) > 0
		char_arg1 = moving average sequence per dimension
		char_arg2 = average type
		= lin: linear
		exp: exponential
		int_arg3 = running length for average
		int_arg3 > 1
		int_arg3 = 0:
		automatic determination:
		= max(3, ext(arg4,i)/20.
				`move_avg('001',`
				`'lin',0,atmo)`

nr_of_runs	number of single runs in the experiment	(6)		`nr_of_runs()`

rank(char_arg1,
arg2)
	assign rank numbers to arg2 according to ranking type argument char_arg1	(1)
		dim(arg2) > 0
		arg1 = ranking type
		[tie_plain | tie_min |
		tie_avg]
				`rank('tie_avg',`
				`atmo)`

regrid(char_arg1,
arg2)
	assign completely or partially new coordinates to arg2	(1), but
		coordinates according to char_arg1
		char_arg1 = file how to transform coordinates of arg2
		arg2 result to transform coordinates
				`regrid('mod.trf',`
				`atmo_g-13)`

run(char_arg1,
arg2)
	values of arg 2 for the selected single run number explicitly or implicitly coded in char_arg1	(1)
		char_arg1 = run number selection
		= 0 for default run (all experiment types)
		= <run_number>
		(for Monte Carlo anal. and loc. sensit. anal.:
		0 ≤ char_arg1 ≤ number_of_runs)
		= <filter argument>
		(for behavioural anal.: same as filter argument of operator behav, check Section 8.4.2)
				`run('0',atmo)`
				`run('sel_t(p1(4))`
				`',atmo)`

Name	Meaning	Argument restriction(s) / result description (see Tab. 8.1)	Argument value restriction	Example

transpose(char_arg1, arg2)	transpose arg2 according to sequence in char_arg1	dim(arg2) > 1
		dim(res) = dim(arg2)
		ext(res,i) = ext(arg2,j) (re-sorted)
		char_arg1 = transpose sequence
				`transpose ('3142',atmo)`

undef( )	undefined value	(6)		`undef()`

Tab. 8.8 *Built-in advanced operators*

The following explanations hold for the operators in Tab. 8.8:

- **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 the values of an operand arg4 that has dimensionality > 0 into the class numbers 1 ,..., int_arg1 of int_arg1 classes. Classes are assumed to be equidistant.
  If both arguments real_arg2 and real_arg3 are 0. then min(arg4) forms the lower boundary of class number 1 and max(arg4) forms the upper boundary of class number int_arg1. For min(arg4) = max(arg4) all result values of the operator classify are undefined.
  For real_arg2 ≠ 0. or real_arg3 ≠ 0 real_arg2 and real_arg3 are used as boundaries for the classification and all of those result values are undefined where values of argument arg4 are outside the specified boundary range.

- The **operator clip** clips an operand arg2 that has dimensionality > 0. The portion to clip from the operand arg2 is described by the argument char_arg1. The argument char_arg1 uses syntax for model output variable addressing (see Section 8.1.3 on page 65). Note, that for all dimensions of argument arg2 lower bound index is 1. This applies also to model output 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 char_arg1 = '* ,..., *' results for operator clip in the identity of argument arg2.

- The **operator cumul** cumulates an operand arg2 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 char_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',arg2) results in the identity to arg2.

- 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 char_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. flip('0...0',arg2) results in the identity to arg2.

- With the **operator get_data** data from external files can be included in post-processing. Character argument char_arg1 specifies the file type. Character argument char_arg2 addresses the data file. Character argument char_arg3 is used to define or transform structure information and coordinates from the data file. Argument arg4 holds the variable that is to be extracted from the data file. For restrictions in the path to the directory of the character arguments char_arg2 and char_arg3 check Tab. 11.3.
  Currently, only ASCII files are supported (char_arg1 = 'ascii'). For ASCII data files the file syntax rules from Section 11.3 are valid. Since the ASCII data file itself does not come with any structure and coordinate information the character agument char_arg3 specifies this information. It follows the same rules as for any coordinate transformation file in Section 11.2. Keywords 'general', 'assign', and 'coordinate' and the appropriate sub-keywords from Tab. 11.5 can be used to structure the data file and to assign coordi-

nates and coordinate values. Consequently, the keyword ‘modify’ is not allowed. See the example below for more information. For ASCII files it is assumed that the file holds only the values for one variable in a sequence according to the Fortran storage model (see Section 15.7 – Glossary). For ASCII files argument arg4 is only a dummy placeholder.

Having a model output variable definition as in Example 5.1 on page 27 and assuming

a data file data.asc as
```
# data file with 6 values
10 , 20 , 30
40 , 50 , 60
```

and a file to define data structure and coordinates data.def as
```
general    descr            structure data.asc
# assign as second dimension coordinate time
# (already defined in world_*.mdf)
assign     2                coord             time
assign     2                coord_extent      11:13
# assign as first dimension a new coordinate new_coord
assign     1                coord             new_coord
assign     1                coord_extent      100:110
coordinate new_coord        values            list 100,110
```

then
```
get_data(‘ascii’,‘data.asc’,‘data.def’,dummy)
```
It has Dimensionality =2
Coordinates = new_coord , time
Extents = 2 , 3

and the result of this operator is a 2 x 3 matrix
```
10   30   50
20   40   60
```

To get same dimensionality, coordinates and extents but result values as the “original matrix” in data.asc
- exchange coordinate numbers in data.def: 1 by 2 and 2 by 1 and
- apply `transpose(‘21’,get_data(‘ascii’,‘data.asc’,‘data.def’,dummy))`

It has Dimensionality =2
Coordinates = new_coord , time
Extents = 2 , 3

and the result of this operator chain is a 2 x 3 matrix
```
10   20   30
40   50   60
```

Example 8.4 *Experiment post-processing operator get_data and coordinate transformation file*

- The **operator get_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 output variables can differ from that used for the current model. Use for the experiment directory char_arg1 always that workspace the external experiment was started from. The external experiment is always post-processed completely over all single runs. Argument char_arg3 is the coordinate transformation file. It can be used to transform coordinates from the external result for usage in the current result of the current experiment. If no coordinate transformation file is to be used argument char_arg3 is empty (‘ ‘). If after potential application of a coordinate transformation file the imported result 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. For syntax of coordinate transformation files check Section 11.2. For restrictions in the path to the directory of the character arguments char_arg1 and char_arg3 check Tab. 11.3.

Attention:

Make sure

- no SimEnv service is running from the directory char_arg1 of the external experiment before applying this operator
- to have full access permissions to the experiment directory char_arg1
- the experiment directory char_arg1 differs from the current workspace

In the experiment directory a file simenv_get_experiment.exc is used to exchange information between the external and the current experiment.

- With the **operator get_table_fct** a table function char_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 char_arg1 has to hold the table function and must be an ASCII file with two columns: The first column of each line is the argument value x associated with the elements of the operand arg2, the second column is the function value f(x) of the table associated with the elements of the operator result. Argument values x 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 Section 11.3). For restrictions in the path to the directory of the character argument char_arg1 see Tab. 11.3. Check the table function world.dat_tab in the example directory $SE_HOME/../examples of SimEnv for more information.

- 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	(char_arg1 = '<')
	arg2 ≤ 0	(char_arg1 = '<=')
	arg2 > 0	(char_arg1 = '>')
	arg2 ≥ 0	(char_arg1 = '>=')
	arg2 = 0	(char_arg1 = '=')
	arg2 ≠ 0	(char_arg1 = '!=')
	arg2 def	(char_arg1 = 'def')
	arg2 undef	(char_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	(char_arg1 = '<')
	arg2 ≤ arg3	(char_arg1 = '<=')
	arg2 > arg3	(char_arg1 = '>')
	arg2 ≥ arg3	(char_arg1 = '>=')
	arg2 = arg3	(char_arg1 = '=')
	arg2 ≠ arg3	(char_arg1 = '!=')

- The **operator matmul** performs a simple matrix multiplication for 2-dimensional arguments arg1 and arg2.

- The **operator move_avg** performs a moving average operation successively for selected dimensions of the argument arg4.
  For a vector (a_1 , a_2 ,..., a_{len}) the moving average of running length rl is a vector (ma_1 , ma_2 ,..., ma_{len}) with elements

$$ma_i = \frac{1}{\sum\limits_{j=\max(1,i-rl+1)}^{i} w_{ij}} \cdot \sum_{j=\max(1,i-rl+1)}^{i} w_{ij} \cdot a_j$$

where w_{ij} are weights. Value ma_i is averaged from the rl values a_i , a_{i-1} ,..., a_{i-rl+1}. Accordingly, the first rl-1 values ma_1, ma_2 ,..., ma_{rl-1} are averaged from less than rl values.

For the linear moving average the weight is $w_{ij} = 1$ and $\sum\limits_{j=\max(1,i-rl+1)}^{i} w_{ij} = \min(rl,i)$,

for the exponential moving average the weight is $w_{ij} = e^{-\frac{i-j}{rl}}$.

While the moving average is normally applied to time-dependent one-dimensional data vectors the operator move_avg allows processing of multi-dimensional data fields in a general and succesive manner. For example, if arg4 is the three-dimensional variable bios(1:lat,1:lon,1:time) then the linear moving average could be applied to the dimension time successively for all combinations of lat and lon. This means that (lat1 = 1 ,..., lat) * (lon1 = 1 ,..., lon) = lat*lon moving averages will be performed for the vector

(bios(lat1,lon1,1) , bios(lat1,lon1,2) ,..., bios(lat1,lon1,time)).

Afterwards this moving averaged temporary result tmp could be moving averaged for all values of lat: (lon1 = 1 ,..., lon) * (time1 = 1 ,..., time) = lon*time moving averages will be performed for the vector

(tmp(1,lon1,time1) , tmp(2,lon1,time1) ,..., tmp(lat,lon1,time1)).

The operator that allows for this double averaging would have the arguments

move_arg(‘201’ , ’lin’ , 0 , bios) .

The character argument char_arg1 supplies those dimensions that are to be involved in the moving average operation. If the n-th digit of char_arg1 is a digit > 0 then the moving average for dimension n of argument arg4 is performed at position number “digit” (i.e. after performing moving averages for those dimensions that correspond to digits smaller than the current one). If the n-th digit of arg1 is 0 then the moving average for the dimension n of arg4 will not be performed.
Keep in mind that the sequence of moving averages for single coordinates influences the result of the operator.

- 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 the operand arg2 that has dimensionality > 0 into their ranks. Small values get low ranks, large values get high ranks. The smallest rank is 1. Character argument char_arg1 determines how to rank ties, i.e., values of arg2 that are identical or have a maximum absolute difference of 10^{-6}:

Assume an argument arg2 with 6 values		(4., 2., 4., 4., 4., 8.).
Then char_arg1 = ‘tie_plain’	returns ranks	(2 , 1 , 2 , 2 , 2 , 3) four times rank 2; next rank is 3, does not take into account the number of identical values
char_arg1 = ‘tie_min’	returns ranks	(2 , 1 , 2 , 2 , 2 , 6) four times rank 2; next rank is 6, takes into account the number of identical values
char_arg1 = ‘tie_avg’	returns ranks	(3.5 , 1 , 3.5 , 3.5 , 3.5 , 6) four times mean rank 3.5 = (2+3+4+5)/4; next rank is 6, takes into account number of identical values

- The **operator regrid** can be used to assign new coordinates to argument arg2. Character argument char_arg1 is the name of the coordinate transformation file that holds the information how to transform the coordinates. The keyword ‘modify’ and the corresponding sub-keywords are not allowed. For syntax of coordinate transformation files check Section 11.2. For restrictions in the path to the directory of the character arguments char_arg1 check Tab. 11.3.

- The **operator run** selects a single run from the run ensemble. The operator run must not contain experiment specific (multi-run) operators as operands, since these operators may refer to the operator run. Additionally, run must not contain itself as an argument.
  The character argument char_arg1 can hold the run number string explicitly. Explicit run number string in character argument char_arg1 is allowed for Monte Carlo and local sensitivity analyses. Additionally, for behavioural and local sensitivity analysis a run number unequal 0 can be selected implicitly by applying a filter of the corresponding operators (see Sections 8.4.2 and 8.4.4) as char_arg1 of the operator run.
  The file <model>.edf_adj holds the target values to be adjusted to the default values for the current experiment. Run number n corresponds with record number n+1 of this file. Single run number 0 corresponds with the default single run 0. For more information on <model>.edf_adj check Section 6.1 on page 43. For examples see Example 8.6 and Example 8.7.

- 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 char_arg1: It consists of digits 1 ,..., dim(arg2) where the digit sequence corresponds with the re-ordered sequence of the operator result extents.
  A character argument char_arg1 = ‘123...’ results for the operator transpose in the identity of argument arg2.

- The **operator undef** supplies a 0-dimensional result as undefined. This operator can be used as an argument for the if-operator.

- For **examples** of all the described operators check Section 8.3.5.

8.3.5 Examples

Having a model output variable definition as in Example 5.1 on page 27 and
assuming address_default=coordinate in world_*.cfg then in experiment post-processing

```
atmo_g+2*atmo_g                  value of result 3*atmo_g
                                 Dimensionality = 1
                                 Coordinates = time
                                 Extents = 20
sqrt(atmo_g)                     square root of atmo_g
                                 Dimensionality = 1
                                 Coordinates = time
                                 Extents = 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
atmo – get_experiment(‘./other_dir’,‘other_model’,‘ ’,atmo)
                                 Difference for atmo between the current experiment and
                                 another model other_model, located in directory ./other_dir
                                 withoutapplication of an coordinate transformation file
                                 Dimensionality = 4
                                 Coordinates = lat , lon , level , time
                                 Extents = according to definition of atmo in other_model
get_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
```

```
      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
      avg(atmo(*,*,*,19:20))            global all-level mean 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 minima of atmo for the last decade and
                                        level 1
                                        Dimensionality = 1
                                        Coordinates = lat
                                        Extents = 11
      minprop_l('10',atmo(20:-20,*,1,20))
                                        zonal tropical indices of those elements of
                                        atmo for the last decade and level 1 where the minimum is
                                        reached the first time
                                        Dimensionality = 2
                                        Coordinates = lat , index
                                        Extents = 11 , 2
      hgr_l('10','bin_no',8,0.,0.,atmo(20:-20,*,1,20))
                                        zonal tropical histograms with 8 bins of atmo for the
                                        last decade and level 1. Bin bound extremes are deviated
                                        from the values of atmo
                                        Dimensionality = 2
                                        Coordinates = lat , bin_no
                                        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
```

Example file: world.post_adv

Example 8.5 *Experiment post-processing with advanced operators*

8.4 Built-In 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).

8.4.1 Standard Aggregation / Moment Operators

Tab. 8.9 summarises multi-run standard aggregation / moment operators for behavioural analysis, Monte Carlo analysis and optimization. They work on the whole run ensemble (for Monte Carlo analysis and optimization) or parts of it (for certain constellations of behavioural analysis post-processing). They are used with suffix _e for Monte Carlo analysis and optimization and without any suffix for behavioural analysis. For a definition of these operators check Tab. 8.2 on page 70.

Aggregation and moment operator	Argument restriction(s) / result description (see Tab. 8.1)
min(arg)	(1)
max(arg)	
sum(arg)	
avg(arg)	
var(arg)	
avgg(arg)	
avgh(arg)	
avgw(arg1,arg2)	(2.1) arg2 = weight
hgr(char_arg1,int_arg2, real_arg3,real_arg4, arg5) (heuristic probability density function)	dim(res) =dim(arg2)+1 ext(res,dim(res)) = number of bins for char_arg1 = 'bin_no' (bin number): coord(res,dim(res)) = name = bin_no values = equidist_end 1(1) number of bins for char_arg1 = 'bin_mid' (bin mid): coord(res,dim(res)) = name = bin_mid values = equidist_end 1st bin mid (bin width) number of bins char_arg1 see above int_arg2 = number of bins 4 ≤ int_arg2 ≤ number_of_runs or 0: automatic determination = max(4,number_of_runs/10) real_arg3 left bin bound for bin number 1 real_arg4 right bin bound for bin number arg2 real_arg3 = real_arg4 = 0.: determine bounds by min(ens(arg5)) and max(ens(arg5)) min(ens(arg5)) = max(ens(arg5)): all result values are undefined
count(char_arg1,arg2)	(1) arg1 = [all | def | undef]
minprop(arg)	(1) return the run number where the extreme is reached the first time. Processing sequence starts with run number 1.
maxprop(arg)	

Tab. 8.9 Multi-run standard aggregation / moment operators

8.4.2 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 sub-keyword ‘comb’ in the experiment description file.

To show the power of the operator behav the simple experiment layouts as described in Fig. 4.3 on page 14 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 Section 6.2.2 on page 46.
- 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(char_arg1, arg2)	navigation and aggregation in the experiment space for arg2 according to char_arg1	char_arg1= selection / aggregation filter according to Tab. 8.14 dim(res) = dim(arg2) + appended dimensions according to char_arg1	

Tab. 8.10 *Experiment specific operator for behavioural analysis*

Placeholder	Explanation
<filter>	' { <operator$_1$> {, <operator$_2$> ... {, <operator$_n$> } ... } } '
<operator>	[<select_operator> | <aggreg_operator> | <show_operator>]
<select_operator>	sel { _<target_val_type>} (<target_name> { <target_val_range> })
<aggreg_operator>	<aggreg_type> {_<target_val_type>} (<target_name> { <target_val_range> })
<show_operator>	show(<target_name>)
<target_name>	name of the experiment target according to the experiment description file
<target_val_type>	specification how to interpret <target_val_range> i as adjustment **i**ndices (indices always count from 1) v as adjustment **v**alues t as resulting **t**arget values
<target_val_range>	[(<val$_1$> { : <val$_2$> }) | (*)] for <val$_2$> = <nil> : <val$_2$> = <val$_1$> (*) : use all values from <target_name> <val$_i$> = <int_val$_i$> for <target_val_type> = i <val$_i$> = <real_val$_i$> else
<aggreg_type>	an aggregation / moment operator from Tab. 8.9 on page 83. 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 hold for the operator **behav**:

- Generally, by the filter argument arg1 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:
  `behav(' ',atmo_g)`
  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 arg2 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 on page 21) 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 (<model>.edf: specific comb file ...) instead of adjustment definitions (<model.edf>: specific [default | <combination>]) all experiment targets are assumed to be adjusted in parallel.

Having a model output variable definition as in Example 5.1 on page 27 and
assuming address_default = coordinate in world_*.cfg
Assume the experiment layout in Example 6.1 (c) on page 46 and
the corresponding experiment description file (c) from Example 6.1 on page 46
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,*))
                                 mean of 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 result 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 8.6 *Experiment post-processing operator behav for behavioural analysis*

8.4.3 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 on page 83 and supplemented with a suffix _e.

Name	**Meaning**	**Argument restriction(s) / result description (see Tab. 8.1)**	**Argument value restriction**
cnf(real_arg1, arg2)	positive distance of confidence measure from mean avg_e(arg2)	(1) real_arg1 error probability	arg1 = [0.001 | 0.01 | 0.05 | 0.1]
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(real_arg1, arg2)	quantile of arg2	(1) real_arg1 quantile value	0. ≤ arg1 ≤ 100.
reg(arg1,arg2)	linear regression coefficient to forecast arg2 from arg1: arg2 = reg(arg1,arg2)*arg1 + n	(2.1)	

Name	Meaning	Argument restriction(s) / result description (see Tab. 8.1)	Argument value restriction
rng(arg)	range = max_e(arg) - min_e(arg)	(1)	
skw(arg)	skewness (3rd moment)	(1)	
stat_full(real_arg1, real_arg2, real_arg3, real_arg4,arg5)	full 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	arg1, arg2 = [0.001 | 0.01 | 0.05 | 0.1] arg1 < arg2 error probability for confidence distance measure 0. ≤ arg3 < arg4 ≤ 100. quantile values
stat_red(real_arg1, real_arg2,arg3)	reduced basic statistical measures of arg3	dim(res) = dim(arg)+1 ext(res,dim(res)) = 7 coord(res,dim(res)) = name = stat_measure values = equidist_end 1(1)7	arg1, arg2 = [0.001 | 0.01 | 0.05 | 0.1] arg1 < arg2 error probability for confidence distance measure

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

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

- The operators **stat_full** and **stat_red** supply basic statistical measures for their last argument. Both operators are stand-alone operators: They must not be operands of any other operator. Contrary, their last argument can be composed from other non-multi-run operators. To store the statistical measures, dimensionality of both operators is that of their last argument, appended by an additional dimension with an extent of 10 and/or 7. Appended coordinate description is pre-defined by SimEnv (check Tab. 10.9).

  These ten data fields (for operator stat_full) and/or seven data fields (operator stat_red) correspond with the following statistical measures:
  1. Deterministic run (run number 0)
  2. Run ensemble minimum
  3. Run ensemble maximum
  4. Run ensemble mean
  5. Run ensemble variance
  6. Run ensemble positive distance of confidence measure from run ensemble mean for real_arg1
  7. Run ensemble positive distance of confidence measure from run ensemble mean for real_arg2

  Only for operator stat_full:

  8. Run ensemble median
  9. Run ensemble quantile of quantile value real_arg3
  10. Run ensemble quantile of quantile value real_arg4

  The operator stat_red was introduced because computation of the median and quantiles consumes a lot of auxiliary storage space. For the definition of the statistical measures check the corresponding single operators in Tab. 8.9 and Tab. 8.12. Both operators were designed for application of an appropriate visualization technique in result evaluation in future.

Having a model output variable definition as in Example 5.1 on page 27 and
assuming address_default=coordinate in world_*.cfg
Assume the Monte Carlo experiment from Example 6.2 (e) on page 49
then in experiment post-processing

```
avg_e(p1*atmo(*,*,1,19:20))   global run ensemble mean 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 mean 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 result expression
                              Dimensionality 4
                              Coordinates = lat , lon , time
                              Extents = 45 , 90 , 4 , 20
hgr_e('bin_no',0,0.,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. Bin bound extremes are
                              derived from the values of the last argument of the operator
                              hgr_e.
                              Dimensionality = 2
                              Coordinates = lat , bin_no
                              Extents = 11 , 25
stat_full(0.01,0.05,25,75, min_l('10',atmo(20:-20,*,1,20)))
                              full 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 8.7 *Experiment post-processing operators for Monte Carlo analysis*

8.4.4 Local Sensitivity Analysis

Tab. 8.13 shows the experiment specific operators for local sensitivity analysis that can be used in post-processing. For a definition of these operators check Tab. 4.3 on page 17.

Name	Meaning	Argument restriction(s) / result description	Argument value restriction
sens_abs(char_arg1, arg2)	absolute sensitivity measure for arg2 according to char_arg1	arg1 = selection / aggregation filter dim(res) = dim(arg2) + appended dimensions according to char_arg1	
sens_rel(char_arg1, arg2)	relative sensitivity measure for arg2 according to char_arg1		
lin_abs(char_arg1, arg2)	absolute linearity measure for arg2 according to char_arg1		
lin_rel(char_arg1, arg2)	relative linearity measure for arg2 according to char_arg1		
sym_abs(char_arg1, arg2)	absolute symmetry measure for arg2 according to char_arg1		
sym_rel(char_arg1, arg2)	relative symmetry measure for arg2 according to char_arg1		

Tab. 8.13 *Experiment specific operators for local sensitivity analysis*

Placeholder	Explanation
<filter>	‘ { <select_operator$_1$> {, <select_operator$_2$> ... {, <select_operator$_3$> } ... } } ‘

<select_operator>	[selt | seli | sels] { _<val_type>} (<val_range>)
	with selt = select target range
	seli = select increment range
	sels = select sign range (only for sens_abs and sens_rel)

<val_type>	specification how to interpret <val_range> i as position **i**ndices (always count from 1) for selt and seli v as increment **v**alues for seli n as target **n**ames for selt n as signs for sels
<val_range>	[(<val$_1$> { : <val$_2$> }) | (*)] for <val$_2$> = <nil> : <val$_2$> = <val$_1$> (*) : use all values from <target_name> <val$_i$> = <int_val$_i$> for <val_type> = i <val$_i$> = <real_val$_i$> for <val_type> = v <val$_i$> = <target$_i$> for <val_type> = n (selt) <val$_1$> = [+ | -] and <val$_2$> = <nil> for <val_type> = n (sels)

Tab. 8.14 *Syntax of the filter argument 1 for local sensitivity operators*

The following rules hold for the filter argument in local sensitivity operators:

- Generally, by the filter argument char_arg1 those runs from the run ensemble are selected (here interpreted as filtered) that are used for the formation of the result.
  Consequently, if no filter is specified all runs are used:

`sens_abs(‘ ’,atmo_g)`
The filter operator has to be specified only if values are to be restricted by corresponding target values, increment values and/or sign ranges.

- For the above three select operators selt, seli and sels the value type is redundant if the value range represents the full range of values by [selt | seli | sels] `(*)`:
  `selt(*) = selt_n(*) = selt_i(*)` and all are redundant.
- Each select operator can be applied only once within the filter argument.
- For <val_type> = i, i.e. if a value range is specified by position indices those targets are selected for selt and/or those increments are selected for seli that correspond with the specified position indices. Position indices are assigned from index 1 to the targets and or increments according to their specification sequence in the corresponding experiment description file <model>.edf.
- If more than one target, increment value and/or sign was selected by the filter argument arg1 it contributes with an additional dimension to the result of the local sensitivity operator:

• For targets	an additional dimension	target_sequ
• For increments	an additional dimension	incr
• For signs	an additional dimension	sign

  is appended to the dimensions of the argument arg2 to form the result of the local sensitivity operator. The extent of this additional dimension corresponds with the defined and/or selected number of targets, increment values and/or signs. For a definition of the additional dimensions check Tab. 10.9.
  Firstly, dimension target_sequ is appended on demand, secondly dimension incr and thirdly dimension sign.

Having a model output variable definition as in Example 5.1 on page 27 and
assuming address_default=coordinate in <model>.cfg
Assume the experiment description file (f) from Example 6.3 on page 50
then in result-processing

```
sens_abs(‘ ’,atmo_g)              absolute sensitivity measure for atmo_g
                                  for all targets, increments and signs
                                  Dimensionality = 4
                                  Coordinates = time , target_sequ , incr , sign
                                  Extents = 20 , 3 , 4 , 2
sens_rel(‘sels_n(+),selt_i(1)’,atmo_g)
                                  relative sensitivity measure for atmo_g
                                  for target p1 and all positive increments
                                  Dimensionality = 2
                                  Coordinates = time , incr
                                  Extents = 20 , 4
sens_abs(‘seli_v(0.001:0.05)’,atmo_g)
                                  absolute sensitivity measure for atmo_g
                                  for all targets, increment values 1 to 3 and all signs
                                  Dimensionality = 4
                                  Coordinates = time , target_sequ , incr , sign
                                  Extents = 20 , 3 , 3 , 2
lin_abs(‘seli_v(0.001:0.05)’,atmo_g)
                                  absolute linearity measure for atmo_g
                                  for all targets and increment values 1 to 3
                                  Dimensionality = 3
                                  Coordinates = time , target_sequ , incr , sign
                                  Extents = 20 , 3 , 3
```

Example file: world.post_f

Example 8.8 *Experiment post-processing operators for local sensitivity analysis*

8.4.5 Optimization

The goal of an optimization experiment is to minimize a cost function by determining the corresponding optimal point in the target space. Nevertheless, the specified model output from all single runs is stored during the experiment.

While the single run that corresponds with the optimal cost function can be post-processed in the single-run modus, the whole experiment can be post-processed as a Monte Carlo analysis. Keep in mind that the targets do not follow a pre-defined distribution.

8.5 User-Defined and Composed Operators / Operator Interface

Besides application of built-in operators during experiment post-processing SimEnv enables construction and application of user-defined and composed post-processing operators. A user-defined operator is supplied by the user in the form of a stand-alone executable that is to perform the operator. Contrarily, a composed operator can be derived from both built-in and user-defined operators to generate more complex operators. User-defined and composed operators are announced to the environment in a user-defined operator description file <model>.odf by their names and the number of character, integer constant, real constant and "normal" arguments. This information is used to check user-defined and composed operators syntactically during experiment post-processing and by the SimEnv service simenv.chk. Sequence of the operator arguments types follows the same rule as for built-in operator (see Section 8.1.4).

A user-defined operator itself is a stand-alone executable that is executed during the check and the computation of the operator chain. While the main program of this executable is made available by SimEnv the user has to supply two functions in C/C++ or Fortran with pre-defined names that represent the check and the computational part. For declaration of both functions SimEnv comes with a set of operator interface functions. They can be used among others to get dimensionality, length, extents and coordinates of an argument and to get and check argument values and to put operator results.

For a composed operator the operator description file <model>.odf simply holds the definition of the corresponding operator chain composed from built-in and user-defined operators and using formal arguments.

8.5.1 Declaration of User-Defined Operator Dynamics

User-defined operators consist of a declarative and a computational part, that are described in one source file in two C/C++ or Fortran functions (see Tab. 8.15):

- Function simenv_check_user_def_operator
  This is the declarative part of the operator. The consistency of the non-character operands can be checked with respect to dimensionality, dimensions and coordinates as well as the values of character arguments can be checked. Dimensionality, extents and coordinates of the result have to be defined, normally in dependence on the argument information.
- Function simenv_compute_user_defined_operator
  This is the computational part of the operator. In the computational part the result of the operator in dependency of its operands is computed.

Function name	Function description	Inputs / outputs / function value	Inputs / outputs / function value description
Functions to host the declarative and computational part in usr_opr_<opr>.[f | c | cpp]			
simenv_ check_user_ def_operator ( )	check consistency of operator arguments and define dimensionality and dimensions of the result	integer*4 simenv_ check_user_ def_operator (function value)	return code = 0 ok ≠ 0 inconsistency between operands
simenv_ compute_user_ def_operator (res)	compute result of the operator in dependency on operands	real*4 res(1) (output)	result vector of the operator
		integer*4 simenv_ compute_user_ def_operator (function value)	return code = 0 ok ≠ 0 user-defined interrupt of calculation **Operator results of a dimensionality > 1 have to be stored to the field res using the Fortran storage model (see Section 15.7 - Glossary).**

Tab. 8.15 *Operator interface functions for the declarative and computational part*

A function value ≠ 0 of simenv_check_user_def_operator( ) should be set according to the following rules:

- If appropriate, forward function value from the operator interface function simenv_chk_2args_[f | c] (see below) to the function value of simenv_check_user_def_operator( ). The corresponding error message is reported automatically by the experiment post-processor. Return code 4 from simenv_chk_2args_[f | c] is only an information and no warning and is not reported.
- Other detected inconsistencies between operands have to be reported to the user by a simple print-statement within simenv_check_user_def_operator. The corresponding return code has to be greather than 5.

Tab. 8.16 summarizes these SimEnv operator interface functions that can be applied in the declarative and computational part written in Fortran or C/C++ (postfix f for Fortran, c for C/C++) to get and put structure information. In this table the input and output data types are documented for functions used in Fortran. For C/C++ the corresponding data types are valid. Implementation of the functions for C/C++ is based on a call by reference for the function arguments.

Function name	**Function description**	**Inputs / outputs / function value**	**Inputs / outputs / function value Description**
Functions to get and put structure information in the declarative and computational part			
simenv_ get_char_arg_ [f | c] (iarg, char)	get string and string length of a character argu-ment	integer*4 iarg (input)	argument number
		character*(*) char (output)	string of the character argument Declare char with a suffficient length.
		integer*4 simenv_ get_char_arg_ [f | c] (function value)	length of character argument
simenv_ get_dim_arg_ [f | c] (iarg, iext)	Iarg > 0: get dimensionality and extents of an argument iarg = 0: get dimensionality and extents of the result	integer*4 iarg (input)	argument number, 0 for result
		integer*4 iext(9) (output)	extents of argument / result iext(1) ... iext(simenv_get_dim_arg_[f | c]...)
		integer*4 simenv_ get_dim_arg_ [f | c] (function value)	dimensionality of argument / result
simenv_ get_len_arg_ [f | c] (iarg)	Iarg > 0: get length of an argument iarg = 0: get length of the result	integer*4 iarg (input)	argument number, 0 for result
		integer*4 simenv_ get_len_arg_f (function value)	length of argument / result
simenv_ get_nr_arg_ [f | c] ( )	get number of arguments of the current operator	integer*4 simenv_ get_nr_arg_ [f | c] (function value)	number of arguments

Function name	**Function description**	**Inputs / outputs / function value**	**Inputs / outputs / function value Description**
simenv_ get_type_arg_ [f | c] (iarg)	Iarg > 0: get data type of an argument iarg = 0: get data type of the result	integer*4 iarg (input)	argument number, 0 for result
		integer*4 simenv_ get_type_arg_f (function value)	type of argument / result = -1 byte = 4 float = -2 short = 8 double = -4 int
simenv_ get_co_chk_ modus_ [f | c] ( )	get level of coordinate check for arguments according to <model>.cfg	integer*4 simenv_ get_co_chk_ modus_ [f | c] (function value)	level of coordinate check for arguments = 0 without = 1 weak = 2 strong
simenv_ get_co_arg_ [f | c] (iarg, ico_nr, ico_beg_pos, co_name)	get formal coordinate numbers and formal coordinate begin value positions of an argument	integer*4 iarg (input)	argument number
		integer*4 ico_nr(9) (output)	formal numbers of the coordinates ico_nr(1) ... ico_nr(simenv_get_dim_ arg_[f | c]...)
		integer*4 ico_beg_pos(9) (output)	formal begin value positions of the coordinates ico_beg_pos(1) ... ico_beg_pos(simenv_get_dim_ arg_[f | c]...)
		character*20 co_name(9) (output)	coordinate names co_name(1) ... co_name(simenv_get_dim_ arg_[f | c]...)
		integer*4 simenv_ get_co_arg_ [f | c] (function value)	return code = 0 ok
simenv_ get_co_val_ [f | c] (ico_nr, ico_pos, co_val)	get for a coordinate a coordinate value at a specified position Application of this function in simenv_check_ user_def_operator for coordinate bin_mid results in an error.	integer*4 ico_nr (input)	formal number of the coordinate (from simenv_get_co_arg_[f | c])
		integer*4 ico_pos (input)	formal position within all coordinate values of the value to get. The smallest ico_pos to use corresponds to the value ico_beg_pos from the function simenv_get_co_arg_[f | c]
		real*4 co_val (output)	coordinate value
		integer*4 simenv_ get_co_arg_ [f | c] (function value)	return code = 0 ok = 1 ico_pos out of range = 2 storage exceeded

Function name	Function description	Inputs / outputs / function value	Inputs / outputs / function value Description
simenv_ chk_2args_ [f | c] (iarg1, iarg2)	check two arguments on same dimensionality, extents and coordinates **If appropriate forward return code ≠ 0 to the function value of simenv_check_ user_def_ operator( )**	integer*4 iarg1 (input)	argument number
		integer*4 iarg2 (input)	argument number
		integer*4 simenv_ chk_2args_ [f | c] (function value)	return code = 0 ok = 1 differing dimensionalities = 2 differing extents = 3 differing coordinates according to the sub-keyword 'coord_check' in <model>.cfg = 4 iarg1 = iarg2
simenv_ put_struct_res_ [f | c] (inplace, idimens {, iext, ico_nr, ico_beg_pos })	put - potential in-place-storage - dimensionality - extents - formal coordinate number - formal coordinate value begin number of the result Currently, only coordinates from the arguments can be assigned to the result. **Has to be applied in the declarative part and only there.**	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 arguments 1, 3 and 5
		integer*4 idimens (input)	dimensionality of the result
		integer*4 iext(9) (input)	only for idimens > 0: extents of the result iext(1) ... iext(idimens)
		integer*4 ico_nr(9) (input)	only for idimens > 0: formal coordinate numbers of the result ico_nr(1) ... ico_nr(idimens)
		integer*4 ico_beg_pos(9) (input)	only for idimens > 0: formal coordinate begin position for formal coordinate number ico_nr of the result ico_beg_pos(1) ... ico_beg_pos(idimens)
		integer*4 simenv_ put_dim_res_ [f | c] (function value)	return code = 0 ok ≠ 0 inconsistency between operands

Tab. 8.16 *Operator interface functions to get and put structural information*

All of these operator interface functions return -999 as an error indicator if an argument iarg is invalid. Output arguments are set to -999 as well.

Tab. 8.17 summarizes these SimEnv operator interface functions that can be applied in the computational part written in Fortran or C/C++ (postfix f for Fortran, c for C/C++) to get and check argument values and put results. In this table the input and output data types are documented for functions used in Fortran. For C/C++ the corresponding data types are valid. Implementation of the functions for C/C++ is based on a call by reference for the function arguments.
To handle real*4 underflow and overflow during computation of the operator results with real*4 argument values it is advisible to compute operator results temporarily as real*8 values and afterwards to transform these values back to the final real*4 operator result by the function simenv_clip_undef_[f | c].

Function name	Function description	Inputs / outputs / function value	Inputs / outputs / function value Description
Functions to get and check argument values and to put results in the computational part			
simenv_ get_arg_ [f | c] (iarg,index)	get value of a non-character argument with index index	integer*4 iarg (input)	argument number
		integer*4 index (input)	vector index of an argument
		real*4 simenv_ get_arg_ [f | c] (function value)	value of argument iarg at index index **Operands of any type are transferred by simenv_get_arg_[f | c] to a real*4 / float representation.** **Operands of a dimensionality > 1 are forwarded to user-defined operators as one-dimensional vectors, using the Fortran storage model (see Section 15.7 - Glossary). Adjust the second argument of simenv_get_arg_[f | c] (index) accordingly.**
simenv_ clip_undef_ [f | c] (value)	**overflow:** set a real*8 value to an undefined real*4 result if appropriate **underflow:** set a real*8 value to real*4 0. if appropriate	real*8 value (input)	value to be checked

		real*4 simenv_ clip_undef_ [f | c] (function value)	Example:
			`res(i)=simenv_clip_undef_[ f | c ]`
			`(value)`

simenv_ chk_undef_ [f | c] (value)	check whether value is undefined before processing it	real*4 value (input)	argument value to be checked
		integer*4 simenv_ is_undef_ [f | c] (function value)	= 0 value is defined = 1 value is undefined
simenv_ put_undef_ [f | c] ( )	set a result value as undefined	real*4 simenv_ put_undef_ [f | c] (function value)	Example: `res(i)=simenv_put_undef_[ f | c ] ( )`

Tab. 8.17 Operator interface functions to get / check / put arguments and results

- In SimEnv the declarative and computational part of a user-defined operator <opr> is hosted in a source file usr_opr_<opr>.[f | c | cpp]. The assigned executable has the name <opr>.opr and has to be located in that directory that is stated in <model>.cfg as the hosting directory opr_directory for user-defined operators.
- The include file simenv_opr_f.inc and simenv_opr_c.inc from the SimEnv home directory can be used in user-defined operators to declare the SimEnv operator interface functions for Fortran and/or C/C++ (see also Tab. 10.4).
- Apply the shell script

  simenv_opr_[f | c | cpp].lnk <opr>

  from the SimEnv home directory to compile and link from usr_opr_<opr>.[f | c | cpp] an executable <opr>.opr that represents the user-defined operator <opr>. Like the main program for the operator also the object $SE_HOME/simenv_opr.o is supplied by SimEnv. This object file has to be linked with usr_opr_<opr>.o and the object library $SE_HOME/libsimenv.a.

- Tab. 15.13 lists the additionally used symbols when linking a user-defined operator.
- In Section 15.3 on page 157 implementation of the user-defined operator matmul_[f | c] is described in detail. It corresponds to the built-in operator matmul. Additionally, check the user-defined operators from Tab. 15.6 and apply them during experiment post-processing.

8.5.2 Undefined Results in User-Defined Operators

Check always by the SimEnv operator interface function simenv_chk_undef(val) (see Tab. 8.17) whether an argument value val is undefined before it is processed.

Set a result to be undefined by the SimEnv operator interface function simenv_put_undef( ) (see Tab. 8.17)
Check usr_opr_matmul_[f | c].[f | c] in Section 15.3 or usr_opr_div.f in the example directory $SE_HOME/../examples of SimEnv for more detailed examples.

If things go so wrong that computation of the whole result expression has to be stopped it is possible to alternatively

- Set all elements of the results to be undefined
- Set simenv_compute_user_def_operator ≠ 0 (otherwise set it always = 0)
- In both cases application of the following operators in the operator chain of the result expression will be suppressed and consequently computation of the result expression will be stopped
- Check usr_opr_char_test.f for a detailed example

8.5.3 Composed Operators

A composed operator is an operator chain composed from built-in and user-defined operators. The concept of composed operators enables construction of more complex operators from built-in and user-defined ones. A composed operator is defined with formal arguments that are used in the operator chain as arguments. Formal arguments are replaced by current arguments when applying a composed operator during experiment post-processing. In this sense, the definition of a composed operator in SimEnv corresponds with the definition of a function in a programming language: When calling the function formal arguments are replaced by current arguments. Consequently, composed operators offer the same flexibility as built-in or user-defined operators.

Like built-in and user-defined operators, a composed operator can have nine formal arguments at maximum. Sequence of these arguments is also the same as for the other operators: Character arguments followed by integer constant arguments, real constant arguments and normal arguments.

For composed operators the operand set (see Section 8.1.2) to form the operator by a chain of operators is restricted to

- Constants in integer and real / float notation
- Character strings
- Operator results from built-in and user-defined operators

Not allowed as operands are

- Model output variables
- Experiment targets
- Composed operators
- Macros

Additionally have to be used

- Formal arguments arg1 ,…, arg9

Check the following Example how to specify composed operators.

```
composed          character        "normal"         composed operator
operator name     argument         argument         definition
-----------------------------------------------------------------------------------------
rel_count  (      arg1       ,     arg2  )    =    100 * count(arg1,arg2) /
                                                   count('all',arg2)
error_1    (      arg1       ,     arg2  )    =    count(arg1,arg2) *
                                                   hgr(arg1,0,0.,0.,arg2)
error_2    (                       arg1  )    =    arg1 *
                                                   hgr('bin_mid',10,0.,0.,arg1)
```

Having a model output variable definition as in Example 5.1 on page 27
then for example, the operator rel_count can be applied by

```
rel_count('def',bios)
rel_count('def',bios(c=20:-20,*,1))
rel_count('undef',100*bios)
```

Example 8.9 *Composed operators*

Composed operators are checked syntactically by the SimEnv service simenv.chk. When performing simenv.chk validity of the following information is **not** cross-checked between formal arguments:

- Character arguments of operators
  Example: The composed operator error_1 is considered by simenv.chk to be valid though
  argument 1 of operator count is limited to values ['all' | 'def' | 'undef'] and
  argument 1 of operator hgr is limited to values ['bin_no' | 'bin_mid']
- Use of "normal" formal arguments in the operator chain with respect to their dimensionality, extents and coordinates
  Example: The composed operator error_2 in is considered by simenv.chk to be valid though
  the dimensionality of the operator hgr in this constellation is always higher than that of the argument arg1 and consequently, multiplication between arg1 and hgr(.) is impossible.

8.5.4 Operator Description File <model>.odf

<model>.odf is an ASCII file that follows the coding rules in Section 11.1 on page 119 with the keywords, names, sub-keywords, and values as in Tab. 8.18. <model>.odf announces the user-defined and composed operators by their names, and the number of character, integer constant, real constant, and normal arguments that belong to an operator. Additionally, <model>.odf hosts for composed operators the corresponding operator chain using formal arguments. <model>.odf is expoited to check a user-defined and/or composed operator syntactically when performing it during experiment post-processing.

keyword	**name**	**sub-keyword**	**Line type**	**Max. line nmb.**	**values**	**Explanation**
general	<nil>	descr	o	any	<string>	general operator descriptions
opr_ defined	<user_ defined_ operator_ name>	descr	o	1	<string>	operator description
		arguments	m	1	<int_val$_1$>, <int_val$_2$>, <int_val$_3$>, <int_val$_4$>	number of arguments defined for the operator: <int_val$_1$> ≥ 0: character arguments <int_val$_2$> ≥ 0: integer constant arguments <int_val$_3$> ≥ 0: real constant arguments <int_val$_4$> > 0: "normal" arguments

keyword	name	sub-keyword	Line type	Max. line nmb.	values	Explanation
opr_ composed	<composed_ operator_ name>	descr	o	1	<string>	operator description
		arguments	m	1	<int_val$_1$>, <int_val$_2$>, <int_val$_3$>, <int_val$_4$>	number of arguments defined for the operator. Explanations and restrictions are the same as for a user-defined operator
		define	m	≥ 1	<string>	operator definition string Operator definition can be arranged at a series of define-lines in analogy to the rules for result expressions (see Section 8.1.1).

Tab. 8.18 *Elements of an operator description file <model>.odf*

To Tab. 8.18 the following additional rules and explanations apply:

- For the description of **line type** check Tab. 11.4 on page 121.
- The sequence of the four integer values <int_val$_1$> ,…, <int_val$_4$> follows the sequence of arguments in built-in, user-defined and composed operators.
- The sum <int_val$_1$> +…+ <int_val$_4$> has to be less equal 9.
- Use the SimEnv service simenv.chk to check user-defined and composed operators.

```
general                      descr      Operator description for the
general                      descr      examples in the SimEnv User Guide

opr_defined   matmul_f     descr      matrix multiplication (in Fortran)
opr_defined   matmul_f     arguments  0,0,0,2

opr_defined   matmul_c     descr      matrix multiplication (in C)
opr_defined   matmul_c     arguments  0,0,0,2

opr_defined   corr_coeff   descr      correlation coefficient r
opr_defined   corr_coeff   arguments  0,0,0,2

opr_defined   div          descr      arithmetic division
opr_defined   div          arguments  0,0,0,2

opr_defined   simple_div   descr      division without undefined-check
opr_defined   simple_div   arguments  0,0,0,2

opr_defined   char_test    descr      test character arguments
opr_defined   char_test    arguments  2,0,0,1

opr_composed  rel_count    descr      relative count [%]
opr_composed  rel_count    arguments  1,0,0,1
opr_composed  rel_count    define     100*count(arg1,arg2)/
opr_composed  rel_count    define     count('all',arg2)
```

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

Example 8.10 *Operator description file <model>.odf*

8.6 Undefined Results

By performing operator chains and due to possibly unwritten model output during simulation parts of the intermediate and/or final result values can be undefined within the float data representation.

If an operand is completely undefined the computation of the result is stopped without evaluating the following operands and operators.

For undefined / nodata value representation check Tab. 10.12.

8.7 Macros and Macro Definition File <model>.mac

- In experiment post-processing a macro is an abbreviation for a result expression, consisting of an operator chain applied on operands.
- Generally, they are model related and they are defined by the user.
- Macros are identified in experiment post-processing expressions by the suffix _m.
- A macro is plugged into a result expression by putting it into parentheses during parsing:
  Example: `equ_100yrs_m*test_mac_m`
  from Example 8.11 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. 15.10)
  - Integer or real constant arguments of built-in operators (check Tab. 15.11)
  - Character arguments of user-defined operators
  - Operators with respect to dimensionality and dimensions of its operands

In SimEnv macros are defined in the file <model>.mac. <model>.mac is an ASCII file that follows the coding rules in Section 11.1 on page 119 with the keywords, names, sub-keywords, and values as in Tab. 8.19. <model>.mac describes the user-defined macros.

keyword	name	sub-keyword	Line type	Max. line nmb.	value	Explanation
general	<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	≥ 1	<string>	macro definition string macro definition can be arranged at a series of define-lines in analogy to the rules for result expressions (see Section 8.1.1).

Tab. 8.19 *Elements of a macro description file <model>.mac*

To Tab. 8.19 the following additional rules and explanations apply:
- For the description of **line type** check Tab. 11.4 on page 121.
- Values for sub-keywords ‘descr’ and ‘unit’ are not evaluated during parsing a result expression.

```
general                 descr      Macro definitions for the
general                 descr      examples in the SimEnv User 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 8.11 *User-defined macro definition file <model>.mac*

8.8 Wildcard Operands &v& and &t&

In SimEnv, wildcard operands offer a convenient approach to compute a result expression successively for all defined model output variables and experiment targets. Wildcard operands are used in the same manner as normal operands when defining a result expression. There are two wildcard operands at disposal:

&v&	wildcard operand for any model output **v**ariable
&t&	wildcard operand for any experiment **t**arget

When applying in a result expression only one wildcard type (i.e., either &v& or &t&) the result expression is performed repetitively where the wildcard is replaced successively by all model output variables and experiment targets, respectively. When applying both &v& and &t& in a result expression the result expression is performed for the Cartesian product of all model output variables and experiment targets.

Wildcard operands must not be used in macro definitions (see Section 8.7). The wildcard operand &v& for model output variables can not be restricted to a portion of the variable by appending a sub-specification in brackets as explained in Section 8.1.3 (e.g., &v&(i=3:10) is not allowed).

Keep in mind that the strings &v& and &t& are only substituted in the result string by model variables and/or model targets if they the strings are

- prefixed by [(| + | - | / | * | begin of result string] and
- postfixed by [(| + | - | / | * | end of result string]

Having a model output variable definition as in Example 5.1 on page 27 and assuming the experiment description file (b) from Example 6.1 on page 46 then in result-processing

```
behav(‘ ’,sin(&v&))         results in
                            behav(‘ ’,sin(atom))
                            behav(‘ ’,sin(bios))
                            behav(‘ ’,sin(atmo_g))
                            behav(‘ ’,sin(bios_g))
behav(‘ ’,&v&*&t&)          results in
                            behav(‘ ’,atmo*p1)
                            behav(‘ ’,bios*p1)
                            behav(‘ ’,atmo_g*p1)
                            behav(‘ ’,bios_g*p1)
                            behav(‘ ’,atmo*p2)
```

```
                                    behav(‘ ’,bios*p2)
                                    behav(‘ ’,atmo_g*p2)
                                    behav(‘ ’,bios_g*p2)
```

Example 8.12 *Experiment post-processing with wildcard operands*

8.9 Saving Results

The result files <model>.res<res_char>.[nc | ieee | ascii] and <model>.inf<res_char>.[ieee | ascii] contain all the model and experiment information for further processing of results.

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.

To get access permission to the SimEnv visualization server use the SimEnv service simenv.key one time. Check Section 10.2 for more information.

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 experiment 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

keyword	name	sub-keyword	Line type	Max. line nmb.	value	Explanation
general	<nil>	descr	o	any	<string>	general configuration description
		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_separation	o	1	[yes | no]	indicates whether to store model output in a single file per single run or in one file per experiment
		auto_interface	o	1	[no | all | f | c | py |sh]	indicates to generate include source code files for the semi-automated model interface for the corresponding languages
		structure	o	1	[standard | distributed | parallel]	indicates model structure with respect to experiment performance
experiment	<nil>	restart_ini	o	1	[no | yes]	perform <model>.ini for experiment re-start
		begin_run	o	1	<int_val>	begin single run number
		end_run	o	1	[last | <int_val>]	end single run number
		email	o	1	<string>	email notification address
postproc	<nil>	out_directory	o	1	<directory>	experiment post-processing output directory
		out_format	o	1	[netcdf | ieee | ascii]	experiment post-processing output format
		address_default	o	1	[coordinate | index]	experiment post-processing address default for model output variables
		coord_check	o	1	[strong | weak | without]	post-processing coordinate check by operators
		opr_directory	o	1	<directory>	directory the post-processors expects user-defined operator executables
		visualization	o	1	[yes | no]	determine whether to directly visualize an entered result during experiment post-processing

Tab. 10.1 *Elements of a 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 Section 11.1 on page 119 with the keywords, names, sub-keywords, and info as in Tab. 10.1.

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

- For the description of **line type** check Tab. 11.4 on page 121.
- **For keyword 'general', sub-keyword 'message_level'**:
  Message output during simenv.chk and to the model interface log-file <model>.mlog is controlled by this information.

Specify	info	to output	errors and warnings and additional information
	warning	to output	errors and warnings
	error	to output	errors

  during simenv.chk and to <model>.mlog.
- **For keyword 'model', sub-keyword 'out_separation'**:
  Specify here whether 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 'auto_interface'**:
  Check Section 5.8.
- **For keyword 'model', sub-keyword 'structure'**:
  Check Section 5.9.
- **For keyword 'experiment', sub-keyword ['begin_run' | 'end_run']:**
  With the exception of an optimization experiment SimEnv enables to perform an experiment partially by performing only an experiment slice out of the whole run ensemble (see Section 7.5 on page 58). There-for assign appropriate run numbers to these two descriptors. Make sure that begin and end run repre-sent run number from the experiment (including run number 0) and that begin run ≤ end run. The value string "last" always represents the last simulation run of the whole run ensemble.
  For an optimization experiment these two sub-keywords are ignored.
- **For keyword 'experiment', sub-keyword 'email'**:
  After performing an experiment an email is sent to the email address specified in <string>.
- **For keyword 'postproc', sub-keyword 'address_default'**:
  During experiment post-processing portions of multi-dimensional model output variables can be ad-dressed by coordinate (c= ...) or index (i= ...) reference. A default is established here.
- **For keyword 'postproc', sub-keyword 'coord_check'**:
  During experiment 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.
- **For keyword 'postproc', sub-keyword 'visualization'**:
  Specifies whether to directly visualize an entered result during experiment post-processing.

Please keep in mind to ensure consistency of control settings in <model>.cfg across different SimEnv ser-vices. As an example one has to run experimentation, experiment post-processing and dump with the same value for out_separation in <model>.cfg.

Tab. 10.2 lists the default values in the general configuration file in the case of absence of the appropriate sub-keyword.

keyword	sub-keyword	Default value (*)	For more information see
general	descr	<nil>	above
	message_level	info	above
model	out_directory	./	above
	out_format	NetCDF	Chapter 12
	out_separation	yes	above
	auto_interface	no	Section 5.8
	structure	standard	Section 5.9 and above
experiment	restart_ini	no	Section 7.3
	begin_run	0	Section 7.1 - 7.5
	end_run	last	Section 7.1 - 7.5
	email	<nil>	Section 7.1
postproc	out_directory	./	above
	out_format	NetCDF	Chapter 12
	address_default	coordinate	Section 8.1.3 and above
	coord_check	strong	Section 8.1.5 and above
	opr_directory	./	Section 8.5
	visualization	yes	above

Tab. 10.2 *Default values for the general configuration file*
***(*)**: in the case of absence of the appropriate sub-keyword*

```
general        descr                  General configuration file for the
general        descr                  examples in the SimEnv User Guide
general        message_level          info

model          out_directory          mod_out
model          out_format             netcdf
model          out_separation         yes
model          auto_interface         f
model          structure              standard

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          ./
postproc       visualization          no
```

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 experiment post-processing there are additional auxiliary SimEnv services to set the SimEnv environment, to check input information consistency, to monitor the status of a running simulation experiment, to dump files of model and experiment post-processor output and to wrap up the SimEnv workspace.

SimEnv service	Use to
Main Services	
simenv.run <model>	prepare and **run** an experiment (see Section 7.1)
simenv.rst <model>	**restart** an experiment (see Section 7.3)

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).
	Before entering experiment post-processing those 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> (<res_char> = latest, default) or with the file number <res_char>. Visualization runs on a remote server.
Auxiliary Services	
simenv.chk <model>	**check** on model script files (<model>.run, <model>.rst, <model>.ini, <model>.end) check <model>.cfg <model>.edf <model>.odf <model>.gdf <model>.mdf <model>.mac existing model and post-processor output files generate pre-experiment output statistics
simenv.sts <model> { <sleep> }	get the current **status** of an active simulation experiment. Start this service from the workspace the active simulation experiment was started from. This is the only service that can be started from a workspace where another service is active.
simenv.dmp <model> <dmp_modus>	**dump** SimEnv model output or experiment 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 storage model for multi-dimensional fields (see Section 15.7 - Glossary).

simenv.cpl
<model>
{ <run> }
{ <file> }
	complete sequence of SimEnv services
	simenv.chk, simenv.run, simenv.res, simenv.vis, simenv.dmp
	simenv.res is performed with input file <file> (if available) and interactively, for both optionally only for single run <run>.

simenv.cln <model>	**clean** up model and experiment post-processor output files Deletes all model output files, post-processor output files, log-files, and auxiliary files of a model according to the settings in <model>.cfg
simenv.cpy <model>	**copy** all SimEnv example files <model>* from the example directory $SE_HOME/../examples 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 if they do not already exist in the current workspace.
simenv.hlp <topics>	acquire basic SimEnv **help** information for the specified topics
simenv.key <user_name>	generate a ssh(2)-**key** to get password-free access to the visualization server. Start this service only one time before the first access to simenv.vis and/or simenv.res or if the ssh(2)-key does not work properly.An email from SimEnv will be sent from SimEnv when the password-free server access is possible.

Tab. 10.3 *SimEnv services*

- With the exception of the simenv.cpy, simenv.hlp and simenv.key:
  Start a service only from the current workspace.
- With the exception of simenv.sts:
  Do not start a SimEnv service from a workspace where an other SimEnv service is still active.

10.3 SimEnv and User Include Files and Link Scripts

In Tab. 10.4 all that include files and link scripts are compiled that are provided by the simulation environment or generated by the user and/or automatically during performing a SimEnv service.

File / location	Used in / generated	Explanation
simenv_mod_ [f | c | cpp].lnk $SE_HOME	used in: stand alone	shell script to compile and link an interfaced model source code for experiment performance If necessary copy to $SE_WS and edit the link stream
simenv_opr_ [f | c | cpp].lnk $SE_HOME	used in: stand alone	shell script to compile and link a user-defined operator source code for experiment post-processing If necessary copy to $SE_WS and edit the link stream
simenv_mod_ [f | c].inc $SE_HOME	used in: interfaced Fortran/C/C++ models	ASCII include file for an interfaced model source code to define SimEnv interface functions and to declare auxiliary variables for the semi-automated model interface
simenv_opr_ [f | c].inc $SE_HOME	used in: interfaced Fortran/C/C++ models	include file for a user-defined operator source code to define SimEnv interface functions
<model>.lnk **(*)** $SE_WS	generated: by the user used in: experiment preparation (only run, not re-start, (only if auto_interface ≠ no in <model>.cfg) and stand alone	shell script to link an interfaced Fortran/C/C++ model. Used in the course of experiment preparation for experiment run (not re-start) if a semi-automated model interface was declared in <model>.cfg for the appropriate programming languages. Can also be used stand alone for non-semi-automated model interface. Is normally based on $SE_HOME/simenv_mod_[f | c | cpp].lnk
<model>_ [f | c | py | sh].inc $SE_WS	generated during: experiment preparation (only run, not re-start, (only if auto_interface ≠ no in <model>.cfg)	ASCII include file for semi-automated model interface The file is to be used directly in the interfaced model source code (for f, c, and py) or in <model>:run as a dot script (for sh)

Tab. 10.4 *SimEnv and user include files and link scripts*
(*)*: make sure by the Unix / Linux command chmod u+x <file>*
that a file <file> has execute permission

Variable	Data type	Used for
simenv_sts	integer*4 / int	SimEnv interface function value
simenv_run_int	integer*4 /int	single run number
simenv_run_char	character*6 / char[6]	6 digit single run number string
simenv_zero	real*4 / float	auxiliary variable, set to 0.

Tab. 10.5 *Contents of $SE_HOME/simenv_mod_[f | c].inc*
(without definition of interface functions)

10.4 User Shell Scripts and Files

Shell script / file (in the current workspace $SE_WS)	Explanation	Exist status	For more information see Section
<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	8.7
<model>.odf	ASCII user-defined operator description file	optional	8.5.4
<model>.gdf	ASCII user-defined GAMS model output description file	for GAMS models mandatory	5.7.2
<model>.run (*)	model shell script to wrap the model executable	mandatory	7.6
<model>.rst (*)	model shell script to prepare single model run restart	optional	7.6
<model>.ini (*)	model shell script to prepare simulation experiment additionally to standard SimEnv preparation	optional, for Python and GAMS models mandatory and standardized	7.6
<model>.end (*)	model shell script to clean up simulation experiment	optional, for GAMS models mandatory and standardized	7.6
<model>.lnk (*)	model shell script to link an interfaced C/C++/Fortran model. Used in the course of experiment preparation for experiment run (not re-start) if a semi-automated model interface was declared in <model>.cfg for the appropriate programming languages	optional	5.8
<model>.jcf_dis	user-specific job control file to submit a job by the load leveler in distributed mode	optional	7.6
<model>.jcf_par	user-specific job control file to submit a job by the load leveler in parallel mode	optional	7.6
<model>.jcf_seq	user-specific job control file to submit a job by the load leveler in sequential mode	optional	7.6
<model>.opt_opt	user-specific control and option file for experiment type optimization	optional	6.5.1
<model>_ <run_char>.err	touch / create this file in the model or in <model>.run as an indicator to stop the complete experiment after <model>.run has been finished for the single model run <run_char>	optional	7.6
<opr>.opr (*) (in the opr_directory according to <model>.cfg)	executable for user-defined operator <opr>	optional	8.5

Tab. 10.6 *User files and shell scripts to perform any SimEnv service*
***(*)**: make sure by the Unix / Linux command chmod u+x <file> that a file <file> has execute permission*

File / location	Generated in	Explanation
Permanent files		
<model>.edf_adj $SE_WS	experiment preparation (all but optimization) experiment performance (optimization)	ASCII adjustment input file for the run ensemble derived from <model>.edf Record no. n+1 corresponds to single run no. n. Column no. m of each record is the adjustment for experiment target no. m in the edf-file
<model>_ [f | c | py | sh].inc $SE_WS	experiment preparation (if auto_interface ≠ no in <model>.cfg)	ASCII include files for semi-automated model interface
<model>.out<run_char> .[nc | ieee] model out_directory	experiment performance (if out_separation = yes in <model>.cfg)	model output of run number <run> of the experiment to be processed by the experiment post-processor
<model>.outall .[nc | ieee] model out_directory	experiment performance (if out_separation = no in <model>.cfg)	model output of all runs of the experiment to be processed by the experiment post-processor
<model>.elog $SE_WS	experiment performance	ASCII minutes file of **e**xperiment performance (simenv.run and all successive simenv.rst)
<model>.mlog $SE_WS	experiment performance	ASCII minutes file of **m**odel interface functions performance (simenv.run and all successive simenv.rst) <model>.mlog is organized single run by single run
<model>.nlog $SE_WS	experiment performance	ASCII minutes file of **n**ative - model specific experim. prepar. by <model>.ini - single runs model output by <model>.run - single run restart preparation by <model>.rst - model specific experim. wrap-up by <model>.end performances, redirected from terminal (simenv.run and all successive simenv.rst) <model>.nlog is organized single run by single run
<model>.res<res_char> .[nc | ieee | ascii] postproc out_directory	experiment post-processing	output file of an experiment post-processing session
<model>.inf<res_char> .[ieee | ascii] postproc out_directory	experiment post-processing	output structure description file of an experiment post-processing session
run<run_char> $SE_WS	experiment performance (only for GAMS models)	sub-directory for GAMS model performance that are kept according to the sub-keyword 'keep_runs' in <model>.gdf
<model>.olog $SE_WS	experiment performance (only for experiment type optimization)	ASCII minutes file of **o**ptimization experiment performance
<model>.edf_cf $SE_WS	experiment performance (only for experiment type optimization)	ASCII file of cost function values. Record no. n+1 corresponds to single run no. n.

File / location	Generated in	Explanation
Temporary files (do not delete during performing the corresponding service)		
<model>. [cfg | mdf | edf | odf | mac]_bin $SE_WS	service dependent	structured binary representation of <model>.[cfg | mdf | edf | odf | mac]
<model>.out<run_char> .[nc | ieee] model out_directory	experiment performance (if out_separation = yes in <model>.cfg)	if the experiment is performed by the load leveler in distributed or parallel mode
<model>.res00.nc $SE_WS	experiment post-processing	NetCDF representation of the current result for visualization during experiment post-processing (only for value “yes” of sub-keyword ‘visualization’ in <model>.cfg)
asa_opt asa_out asa_usr_out $SE_WS	experiment performance (only for experiment type optimization)	auxiliary files for experiment type optimization
run<run_char> sub-direct. of $SE_WS	experiment performance (only for GAMS models)	sub-directory for GAMS model performance that are not kept according to the sub-keyword ‘keep_runs’ in <model>.gdf
<model>_ [pre | main | post].inc $SE_WS	experiment performance (only for GAMS models)	auxiliary files <model> = GAMS main and all interfaced sub-models
simenv_get_experiment .exc $SE_WS	experiment post-processing	auxiliary file for operator get_experiment
simenv_*.tmp $SE_WS	all services	auxiliary files

Tab. 10.7 *Files generated during performance of SimEnv services*
For the current workspace $SE_WS see Tab. 10.13.

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

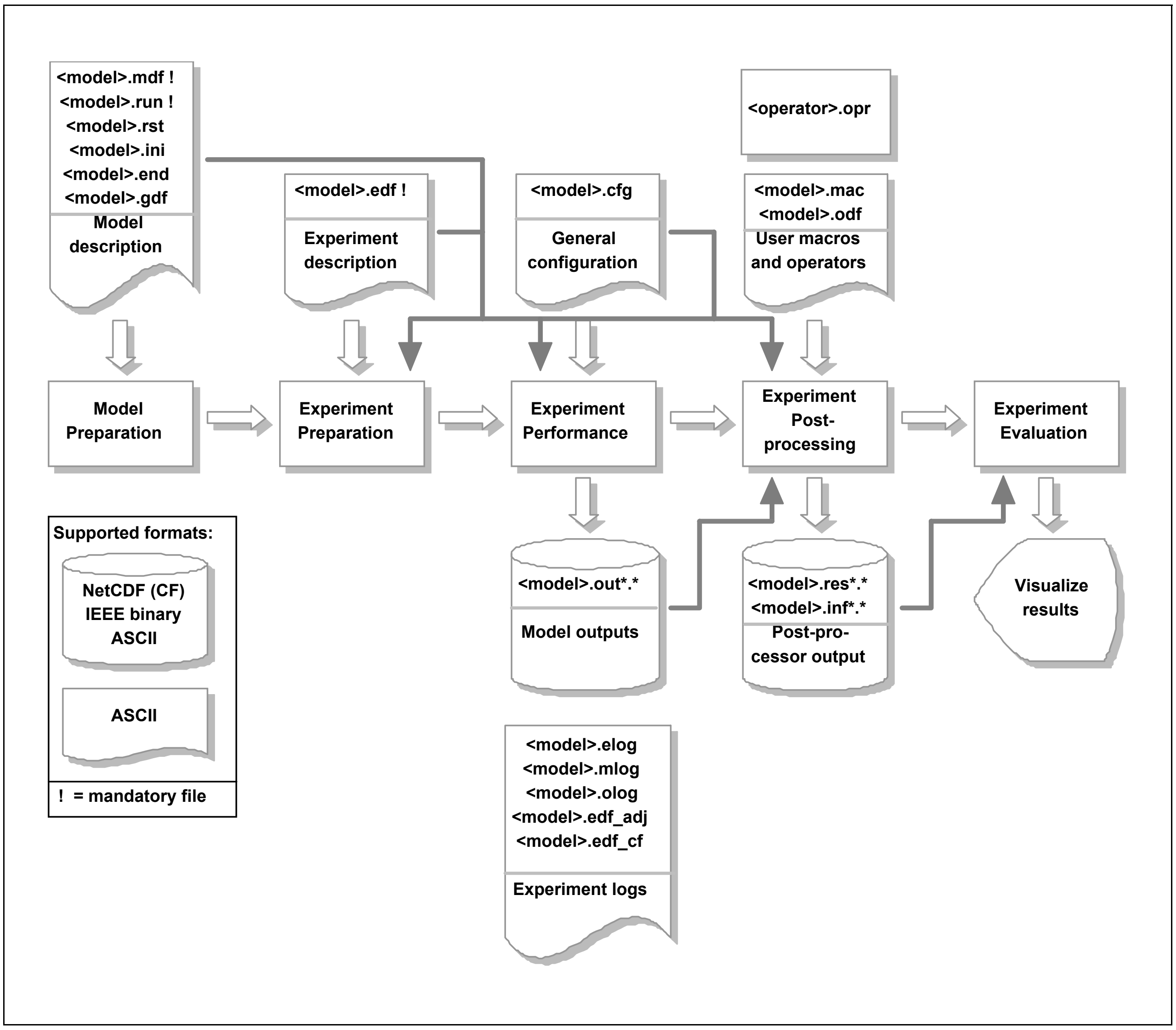


Fig. 10.1 *SimEnv user shell scripts and files*

10.5 Built-In Names

SimEnv has a number of built-in model output variable, coordinate and shell script variable names that can not be used for corresponding user-defined names.

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

Built-in model output variable name	Dimen-sionality	Extents	Data type	Meaning
sim_time	0		float	elapsed simulation time in seconds (rounded to two decimal places) when performing <model>.run

Tab. 10.8 *Built-in model output variables*

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

Built-in coordinate name	Generated by operator	Meaning	Definition (check Tab. 11.6)
bin_mid	hgr, hgr_e, hgr_l	bin mid values	equidist_end <xx>(<yy>) 999999 with <xx> = first bin mid <yy> = bin width
bin_no	hgr, hgr_e, hgr_l	bin numbers	equidist_end 1(1)999999
incr	lin_abs, lin_rel, sens_abs, sens_rel, sym_abs, sym_rel	increment values	dependent on experiment description and operator arguments
index	maxprop, maxprop_l, minprop, minprop_l,	index number	equidist_end 1(1)999999
run	ens	run numbers	equidist_end 1(1)999999
sign	sens_abs, sens_rel	signs of incremental change: -1: -ε +1: +ε	equidist_end -1(2)1
stat_measure	stat_full, stat_red	basic statistical measures: 1: Deterministic case 2: Minimum 3: Maximum 4: Mean 5: Variance 6: Positive distance of confidence measure 1 from mean 7: Positive distance of confidence measure 2 from mean 8: Median 9: Quantile of quantile value 1 10: Quantile of quantile value 2	equidist_end 1(1)10
target_sequ	lin_abs, lin_rel, sens_abs, sens_rel, sym_abs, sym_rel	sequence of targets: 1: 1st target in edf-file 2: 2nd target in edf-file ...	equidist_end 1(1)999999
<target_name>	behav	target values	dependent on experiment description and operator arguments

Tab. 10.9 *Built-in coordinates*

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

Built-in 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

Tab. 10.10 *Built-in shell script variables in <model>.run*

10.6 Case Sensitivity

As stated in Tab. 10.11 all names used in SimEnv are case insensitive. Internally, they are mapped on a lowercase representation and this lowercase representation is used also for model and/or experiment post-processor output files in NetCDF, IEEE and/or ASCII format.

Where?	Entity	Case sensitivity	Example
overall	• model name	sensitive	`simenv.chk World_f`
user-defined files (see Section 11.1)	• keyword • name exception: GAMS model file name in <model>.gdf • sub-keyword	insensitive	`experiment  END_RUN  last`
	• information <value> exceptions: • <directory> and <file_name> - for <sub-keyword> = '<string>_directory' - and in <val_list> • <value> for <sub-keyword> = ['descr' | 'unit'])	insensitive	`experiment  end_run  LAST` `general  descr  This is ...` exception: `specific  comb  file AbC.d`
model interface	• variable and target name	insensitive	`call simenv_put_f('ATMO',atmo)` `target_name='P1'` `target_value=1.` `. $SE_HOME/simenv_get_sh`
experiment post-processing	• optional result description and unit	sensitive	`Energy [kW] = my_opr(atmo)`
	• variable and target name • operator name • number • macro name • macro identifier _m	insensitive	`3e-6*exp(atmo) +` `3E-6*EXP(ATMO)`
	• character arguments of built-in operators with pre-defined values (check Tab. 15.10)	insensitive	`count('ALL' , atmo)`
	• character arguments of built-in operators without pre-defined values	check Tab. 15.10	`get_table_fct('MyFile.dat' , atmo)` `get_experiment('../' , 'Model_f' ,' ', atmo)`
	• character arguments of user-defined operators	sensitive	`char_test('arg11' , 'Arg21' , atmo)`

Tab. 10.11 *Case sensitivity of SimEnv entities*

10.7 Nodata Representation

For model output with the SimEnv model coupling interface functions and for experiment 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
byte	= 127
short	= 32767
int	= 2147483648
float	≥ 3.4E+38
double	≥ 1.79D+308

Tab. 10.12 *Data type related nodata values*

10.8 Environment Variables

The following operating system environment variables are used by SimEnv.
Additionally, make sure that in the shell the noclobber option is not set.

Environment variable	Meaning	Explanation
DISPLAY	machine / screen that the X11-system uses for displaying windows	has to be defined by the user only for visualization matters in SimEnv services simenv.res and simenv.vis: Value = machine dependent Specify also explicitly when logged in at a machine by using a secure socket shell client ssh(2)
SE_HOME	SimEnv home directory	Value = /usr/local/simenv/bin Prefix PATH by $SE_HOME to get any SimEnv service directly
Set inside SimEnv (check $SE_HOME/simenv.env for values)		
PYTHONPATH	path to search Python files	Value = dependent on Python installation Expanded by $SE_HOME
PYTHON_ VERSION	Python version	Value = dependent on Python installation
PYTHON_ ROOT	Python root directory	Value = dependent onPython installation
Set inside SimEnv		
SE_GUI	identificator for GUI / non-GUI version	defined automatically in any SimEnv service Value = [yes | no]
SE_MOD	model name	defined automatically in any SimEnv service Value = <model>
SE_OS	operating system specification	defined automatically in any SimEnv service Value = [AIX | LINUX]
SE_WS	current SimEnv workspace	defined automatically in any SimEnv service Value = <directory>
SE_RUN	run number of a single run	defined automatically in <model>.run and <model>.rst Value = <run_int>
SE_RUN1	first single run of an experiment	defined automatically in <model>.run and <model>.rst Value = [yes | no]

Tab. 10.13 *Environment variables*

11 Structure of User-Defined Files, Coordinate Transformation Files, Value Lists

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. Additionally, coordinate transformation files are described and value lists are defined in general.

11.1 General Structure of User-Defined Files

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> <value> { <sep> }

with

• <name>	is the name of a • model output variable • GAMS model source file • experiment target • coordinate • user-defined operator or • macro Declaration of <name> depends on the related keyword <keyword>
• <keyword>	is a string Normally, more than one lines with differing sub-keywords belong to one "keyword-block".
• <sub-keyword>	is a string Sub-keywords are defined only in relation to the user file and the keyword under consideration.
• <value>	= <substring> { <sep> <substring> ... } is a string with user file, keyword and sub-keyword related information.
• <sep>	is a sequence of white spaces

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 example below. Sequence of names in the separated name spaces (name spaces of coordinates, model output 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.
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 the contents of user-defined files check Tab. 10.11 on page 117.

File	Contents	See description	
		in Section	on page
<model>.cfg	general configuration file	10.1	107
<model>.mdf	model output description file	5.1	21
<model>.gdf	GAMS description file	5.7.2	36
<model>.edf	experiment description file	6.1	43
<model>.odf	operator description file	8.5.4	99
<model>.mac	macro description file	8.7	101
arbitrary file name	coordinate transformation file	11.2	122

Tab. 11.1 *User-defined files with general structure*

The following restrictions hold for 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. 8.3 on page 70)
<value>	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. 100 characters (for the resulting resolved directory string, directory can contain operating system environment variables)
	for sub-keyword = 'unit': max. 32 characters

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

Tab. 11.3 lists the reserved (forbidden) names, file names and directories to files that can not be declared in user-defined files.

Element	Reserved (forbidden) names
<name> excepted for GAMS model source code file names	built-in model output variables according to Tab. 10.8
	built-in coordinates according to Tab. 10.9
	special keywords in <model>.edf for behavioural analysis: [default | file]
<directory>	can contain operating system environment variables ($...) If <directory> is specified in a relative manner it relates to the current workspace.
<file_name>	SimEnv file names according to Tab. 10.6 and Tab. 10.7

Tab. 11.3 *Reserved names and file names in user-defined files*

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

Abbre-viation	User-defined file	Explanation
m	all files	**m**andatory
o	all files	**o**ptional
c1	<model>.mdf keyword 'variable' sub-keyword ['coords' | 'index_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 for Monte Carlo analysis keyword 'target' sub-keyword 'sampling'	**c**onditional **4**: mandatory for adjusts = distr ... forbidden for adjusts = 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. 11.4 *Line types in user-defined files*

```
mac                  descr      This is a macro description file
mac                  descr      for the SimEnv User 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*

11.2 Coordinate Transformation File

Some operators (currently, get_experiment and get_data) enable access to external data. Most of these operators derive in general from an operator argument a multi-dimensional result that has to be equipped - as usual in SimEnv experiment post-processing - with a coordinate assignment. By applying these operators it can be necessary to define or transform a coordinate description for the operator result that fits the result to the current model and/or experiment under consideration. The following cases can be distinguished:

- A dimension of the result does not have a coordinate assignment. A coordinate has to be assigned to this dimension.
- A coordinate description of the result has to be modified in a way that it matches with a defined coordinate of the model / experiment under consideration.
- A coordinate description of the result has to be incorporated with and/or without modifications into the coordinate set of the model / experiment under consideration.

Coordinate transformations for results in the course of the operator's performance are supported by a coordinate transformation file that is assigned to the operator result as an argument of the operator. Coordinate transformation files follow the same syntax rules as all other user-defined files (see Section 10.1).

keyword	name	sub-keyword	Line type	Max. line nmb.	value	Explanation
general	<nil>	descr	o	any	<string>	general transformation description
modify	<original_ coordinate_ name>	rename	o	1	<new_name>	renames original coordinate
		position_shift	o	1	<real_val>	shifts all values of the original coordinate by the specified value <position_shift_val>
		values_shift	o	1	<int_val>	shifts the result values on the original coordinate by the specified positions <values_shift_val>
		values_add	o	1	<val_list>	defines <values_shift_val> values to add to the coordinate values (for syntax see Tab. 11.6)
assign	[<original_ coordinate_ name> | <coordinate_ nmb>]	coord	o	1	<co_name>	assign to the dimension with coordinate number <coordinate_nmb> (only for operator get_data('ascii',…) and/or <original_coordinate_name> (else) an already defined coordinate or a coordinate defined by the keyword 'coordinate'
		coord_extent	o	1	<co_val$_1$>:<co_val$_2$>	assigns start and end coordinate value to the dimension of the result under consideration
coordinate	<new_ coordinate_ name>	descr	o	1	<string>	coordinate axis description
		unit	o	1	<string>	coordinate axis unit
		values	o	1	<val_list>	strictly monotonic sequence of coordinate values (for syntax see Tab. 11.6)

Tab. 11.5 *Elements of a coordinate transformation file*

To Tab. 11.5 the following additional rules and explanations apply:

- For the description of **line type** check Tab. 11.4 on page 121.
- With the sub-keyword '**values_shift**' result values can be shifted on the corresponding coordinate by <values_shift_val> coordinate values. Consequently, <values_shift_val> coordinate values have to be appended at the end of the coordinate for a positive value of <values_shift_val> and/or have to be inserted at the begin of the coordinate for a negative value of <values_shift_val>. Coordinate values that are obsolete because of this shift are removed from the coordinate definition.
  For a coordinate that is defined with equidistant coordinate values the extent of the coordinate is specified automatically by simply applying the equidistant rule for this coordinate.
  For a coordinate with non-equidistant coordinate values the coordinate values necessary for the coordinate extension are defined by the sub-keyword '**values_add**'.
  If both '**position_shift**' and '**values_shift**' are specified for one coordinate, firstly position shift is applied to the coordinate and then the additional coordinate values from values_shift are added the the coordinate without applying the position_shift value.
- Coordinate numbers **<coordinate_nmb>** are integers counting from 1.
- For the sub-keyword '**coord_extent**' the same rules apply as for the sub-keyword 'coord_extents' from the model output description file <model>.mdf.
- For the keyword '**coordinate**' the same rules apply as for the keyword 'coordinate' from the model output description file <model>.mdf.
- Coordinates are incorporated additionally into the original coordinate set only for the current result.

Unlike all other user-defined files coordinate transformation files can not be checked by the SimEnv service simenv.chk or when starting the service simenv.res.

Having a model output variable definition as in Example 5.1 on page 27 and
assuming address_default = coordinate in <model>.cfg
Assume the experiment layout in Example 6.1 (c) on page 46 and
the corresponding experiment description file (c) from Example 6.1 on page 46.

Assume additionally result from another experiment with a model named model and there
a result modvar1+modvar2 that is defined for the following coordinates:

dimension	coordinate name	coordinate definition
1	dim1	list 1,10,100,1000
2	dim2	equidist_end 2(2)20
3	dim3	equidist_end 3(3)30
4	dim4	equidist_end 4(1)43
5	dim5	equidist_end 5(1)50

Further, assume the coordinate transformation file model.trf as

```
general                       descr              example of a coordinate
general                       descr              transformation file

modify       dim1             rename             new1
modify       dim1             position_shift     3.
modify       dim1             values_shift       +2
modify       dim1             values_add         list 1006,1009

modify       dim3             values_shift       -3

assign       dim4             coord              lat
assign       dim4             coord_extent       88.:-68.

assign       dim5             coord              new2
assign       dim5             coord_extent       50.:5.
```

```
coordinate     new2         descr              new coordinate
coordinate     new2         values             equidist_end 50(-1)5
```

In experiment post-processing the result of the expression

```
get_experiment('mydir','model','model.trf',modvar1+modvar2)
```

is a 5-dimensional data structure with

dimension	coordinate name	coordinate definition	coordinate use
1	`new1`	`list 103,1003,1006,1009`	= coordinate definition
2	`dim2`	`equidist_end 2(2)20`	= coordinate definition
3	`dim3`	`equidist_end -6(3)21`	= coordinate definition
4	`lat`	`equidist_end 88(-4)-88`	`equidist_end 88(-4)-68`
5	`new2`	`equidist_end 5(1)50`	= coordinate definition

Example 11.2 *Coordinate transformations by a transformation file*

11.3 Value Lists

For variables, coordinates and experiment targets value lists are supplied by the value-item in user-defined files. Value lists describe a sequence of values together with an order. The number of described values has to be greater than 1. Value lists may be restricted to strictly monotonic sequences. They follow the syntax rules in Tab. 11.6.

Value-list type	Syntax		Explanation
explicit	list	<real_val$_1$> ,..., <real_val$_n$>	explicit list of values same syntax rules as for one record of a file with a value list (see below)
by reference	file	{<directory>/}<file_name>	file {<directory>/}<file_name> contains the explicit value list
implicit with end-element	equidist_end	<real_val$_1$> (<real_val$_2$>) <real_val$_3$>	description of an equidistant list of values with begin value <real_val$_1$> increment <real_val$_2$> end value <real_val$_3$> <real_val$_1$> ≠ <real_val$_3$> <real_val$_2$> ≠ 0.
implicit with number of values	equidist_nmb	<real_val$_1$> (<real_val$_2$>) <int_val>	description of an equidistant list of values with begin value <real_val$_1$> increment <real_val$_2$> number of values <int_val> <real_val$_2$> ≠ 0. <int_val> > 0

Tab. 11.6 *Syntax rules for value lists*

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

- Has to be an ASCII file
- May be a multi-record file
- Max. record length is 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
- Records formed only from white spaces or records starting with the first non-white space character # are handled as comments

1.	`list 3, 5, 7, 9, 11`	describes the five values 3, 5, 7, 9, and 11
2.	`equidist_end 3 (2) 11`	is equivalent to 1.
3.	`equidist_end 3 (2) 11.9`	is equivalent to 1.
4.	`equidist_nmb 3 (2) 5`	is equivalent to 1.

5.	`file my_values.dat`	is equivalent to 1. with `my_values.dat =` `3, , 5,`
		`7`
		`9,`
		
		`11`

6.	`equidist_end 11 (-2) 3`	differs from 1. – 5.: values are identical, ordering sequence differs

Example 11.3 *Examples of value lists*

12 Model and Experiment Post-Processor Output Data Structures

This chapter summarizes information on available data structures for model and experiment 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 experiment 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 files or a common file depends on specification of the value for the sub-keyword 'out_separation' in <model>.cfg. <run_char> is a six-digit placeholder for the corresponding single run number.
During experiment 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 experiment post-processor output formats, multi-dimensional data is organized in the Fortran storage model (see Section 15.7 - Glossary).
Use the SimEnv service command simenv.dmp for browsing model and result output files. See Tab. 10.3 for more information.

12.1 NetCDF Model and Experiment Post-Processor Output

The intention for supplying NetCDF format for model and experiment 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.

In principle, any NetCDF file can be viewed by the NetCDF service program

```
ncdump <NetCDF_file>
```

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 (see Tab. 5.4)	NetCDF data type
byte	NF_BYTE
short	NF_SHORT
int	NF_INT
float	NF_FLOAT
double	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 output description according to <model>.mdf	char
:model_description_file	{<directory>/}<model>.mdf	char
:experiment_type	[behaviour | monte carlo | local sensitivity | optimization]	char
:experiment_description	experiment description according to <model>.edf	char
:experiment_description_file	{<directory>/}<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 output 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>:unit	[<coordinate_unit> | <predef_coordinate_unit> | <predef_variable_unit> | <target_unit> | <variable_unit> | <result_unit>]	char
<variable_name>:missing_value	<variable type-depending missing value>	type-dep.
<variable_name>:axis (single coordinate variables only)	[X | Y | Z | T | bin_no | run | …]	char
<variable_name>:coordinates (multi-dimensional coordinate variables only)	<par1_lon> <par1_lat>	char
<variable_name>:_Fillvalue	<variable type-depending fill value>	type-dep.

Tab. 12.3 *Variable NetCDF attributes*

- For experiment 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 sub-keywords for the keyword 'variable' 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 **:_Fillvalue** 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 output 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 output variable, target, model output 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. For the determination of the record length see below.

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 output variable / target is according to its dimensionality, extents and data type. Storage demand in bytes for each model output 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 storage model (see Section 15.7 - Glossary).
Data is stored in records with a fixed record length of minimum(512000 Bytes , readjusted storage demand in Bytes).
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>. Consequently, 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

One single run needs 1.555.312 : 512.000 = 3+1 records with a fixed length of 512.000 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 Experiment Post-Processor Output

For IEEE and ASCII experiment 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 float / int 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:

result number 01:		
record no. 1	max. 512 chars	result expression string
record no. 2	max. 128 chars	result description string
record no. 3	max. 32 chars	result unit string (or 1 space if unit is undefined)
record no. 4	10 int	dim ext(1) ... ext(dim) 0 ... 0
record no. 4	max. 20 chars	coordinate name of dimension 1
record no. 5	10 float	coordinate values of dimension 1 in records of 10 values (last record may have less values)
...		
record no. xxx	max. 20 chars	coordinate name of dimension dim
record no. xxx+1	10 float	coordinate values of dimension dim in records of 10 values (last record may have less values)
result number 02:		
...		

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

result number 01:		
record no. 1 ...	10 float	in records of 10 values (last record may have less values): result_value(1) ... result_value(length_result)
		with length_result = $\prod_{i=1}^{dim} ext(i)$ for dim > 0
		= 1 else
result number 02:		
...		

The vector result_value is stored in the Fortran storage model (see Section 15.7 - Glossary). The nodata element for undefined result values is set to 3.4E38.

The Fortran code in Example 15.11 reads experiment post-processing ASCII output files <model>.res<res_char>.ascii and <model>.inf<res_char>.ascii in their general structure. In the examples-directory $SE_HOME/../examples of SimEnv it is accompanied by the corresponding version for IEEE result output.

13 SimEnv Prospects

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

General

- Graphical user interface
- Portability to Windows-based systems
- Unique number representations for binary output of distributed models (big endians vs. small endians)

Model interface

- Interface for Java models
- simenv_slice_py for Python models

Experiment preparation

- Experiment type qualitative global sensitivity analysis
- Experiment type uncertainty analysis with variance decomposition
- Experiment type stochastic analysis
- Monte Carlo analysis: stopping rules and sampling of correlated targets

Experiment performance

- Experiment performance for distributed models across networks
- Multi-file model output storage

Experiment post-processing

- Additional advanced operators (coarse, sort, categorical operators)
- Advanced uncertainty and global sensitivity analyses operators
- Flexible assignment of data types to operator results (currently: only float)
- Shared memory access for user-defined operators to avoid data exchange by external files

Experiment evaluation

- Advanced techniques for graphical representation of experiment 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, Joint Research Centre – 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
http://www.pik-potsdam.de/reports/pr-47/pr47.pdf

Flechsig, M., Böhm, U., Nocke, T., Rachimow, C. (2005): Techniques for Quality Assurance of Models in a Multi-Run Simulation Environment. In: Hanson, K.M., Hemez, F.M. (eds.): Sensitivity Analysis of Model Output. Proceedings of the 4th International Conference on Sensitivity Analysis of Model Output (SAMO 2004). Los Alamos National Laboratory, Los Alamos, U.S.A., 297-306
http://library.lanl.gov/ccw/samo2004/

Gray, P., Hart, W., Painton, L., Phillips, C., Trahan, M., Wagner, J. (1997) A Survey of Global Optimization Methods. Sandia National Laboratories, Albuquerque, U.S.A.
http://www.cs.sandia.gov/opt/survey

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

Ingber, L. (1989): Very fast simulated re-annealing. Math. Comput. Modelling, 12(8), 967-973
http://www.ingber.com/asa89_vfsr.pdf

Ingber, L. (1996) Adaptive simulated annealing (ASA): Lessons learned. Control and Cybernetics, 25(1), 33-54
http://www.ingber.com/asa96_lessons.pdf

Ingber, L. (2004) ASA-Readme.
http://www.inger.com/ASA-README.pdf

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

Schulzweida, U. (2004): Climate Data Operators. Max-Planck-Institute for Meteorology, Hamburg and
http://www.mpimet.mpg.de/ ~cdo

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
http://www.pik-potsdam.de/pik_web/publications/pik_reports/sum_pr13.htm

Wenzel, V., Matthäus, E., Flechsig, M. (1990) One Decade of SONCHES. Syst. Anal. Mod. & 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

The appendices summarize the current version implementation, list the examples for model interfaces, user-defined operators and result import interfaces, and they compile all experiment post-processor built-in operators. Finally, a glossary of the main terms as used in this User Guide is supplied.

15.1 Version Implementation

Currently, SimEnv is implemented under Unix and Linux. The Unix version is available at PIK from /usr/local/simenv. The Linux version is directly available from the SimEnv developers. For both versions, only the latest version is supported and bug fixes are installed on demand. Tab. 15.1 lists the directory structure of SimEnv.

Sub-directory of /usr/local/bin	Contents
bin	latest version of SimEnv
doc	documentation for the latest version
examples	example files for the latest version
version_archive	version archive of SimEnv, structured for each version in the same manner as for the latest version

Tab. 15.1 *SimEnv installation under Unix*

15.1.1 System Requirements

Component	Specification	
	Unix	Linux
hardware	RS6000 and compatibles	Intel-based systems and compatibles
operating system	AIX Version 4.3 or higher	SUSE Version 9.0 or higher http://www.suse.com
shell	Bourne shell sh	
Fortran compiler (only for compiling and linking interfaced Fortran models and user-defined operators)	xlf IBM Fortran compiler	ifort Intel Fortran compiler
C/C++ compiler (only for compiling and linking interfaced models and user-defined operators written in C/C++)	xlc IBM C/C++ compiler	gcc GNU C/C++ compiler
Python	Version 2.3 or higher http://www.python.org	
OpenDX (Linux-based visualization server)	Version 4.3.2 or higher http://www.opendx.org	
NetCDF-CF	Version 1.04 or higher http://www.cgd.ucar.edu/cms/eaton/cf-metadata	

Tab. 15.2 *System requirements*

15.1.2 Technical Limitations

Entity		Limitation
User-defined files entities (check also Section 11.1)		
max. length of a record in a user-defined file	[characters]	160
max. length of all general descriptions descr	[characters]	512
max. length of a local description descr	[characters]	128
max. length of a unit	[characters]	32
max. length of a name	[characters]	20
max. sum from user-defined and composed operators in <model>.odf		45
max. length of all define strings for a macro or a composed operator	[characters]	512
max. length of a record of a referred data file	[characters]	1 000
Model interface and experiment preparation entities		
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		50
max. number of coordinates		30
max. number of experiment targets		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
Experiment post-processing entities (per result)		
max. length of the optional result description string	[characters]	128
max. length of the optional result unit string	[characters]	32
max. number of arguments of an operator		9
max. dimensionality of a result		9
max. dimensionality of a result stored in NetCDF format		4
max. length of a complete result string (with description and unit)	[characters]	512
max. number of all operands and operators of a result		200
max. length of a string for a constant	[characters]	20
max. number of constants		30
max. number of allocatable main memory segments		10
max. allocatable main memory	[MBytes]	240
max. number of post-processor output files		99

Tab. 15.3 *Current SimEnv technical limitations*

15.1.3 Linking User Models and User-Defined Operators

- User models implemented in C/C++ or Fortran have to be linked with the following libraries to interface them to the simulation environment
  - $SE_HOME/libsimenv.a
  - /usr/local/lib/libnetcdf.a
- User-defined operators to be used in experiment post-processing have to be linked with the following library to interface them to the simulation environment
  - $SE_HOME/libsimenv.a

For running interfaced models outside SimEnv check Section 5.10.

15.1.4 Example Models and User Files

For the following models corresponding files of Tab. 10.6 of can be copied from the corresponding examples-directory $SE_HOME/../examples to the user's current workspace by running the SimEnv service command simenv.cpy <model> from this workspace:

model	Language / source code	Explanation
world_f	Fortran world_f.f	global atmosphere - biosphere model at resolution of (lat x lon x level x time) = (45 x 90 x 4 x 20)
world_c	C world_c.c	
world_cpp	C++ world_cpp.cpp	
world_py	Python world_py.py	
world_sh	Shell script level world_sh.f world_shput.f	
world_f_auto (semi-automated model interface)	Fortran world_f_auto.f	
world_sh_auto (semi-automated model interface)	Shell script level world_sh.f world_shput.f	
world_f_1x1	Fortran world_f_1x1.f	global atmosphere - biosphere model at a resolution of (lat x lon x level x time) = (180 x 360 x 16 x 20)
world_f_05x05	Fortran world_f_05x05.f	global atmosphere - biosphere model at a resolution of (lat x lon x level x time) = (360 x 720 x 16 x 20)
gridcell_f	Fortran gridcell_f.f	global atmosphere - biosphere model for one lat-lon grid cell at a resolution of (level x time) = (4 x 20)
gams_model	GAMS gams_model.gms	GAMS example model

Tab. 15.4 *Implemented example models for the current version*
For the generic model "world" check Example 1.1

Additionally, the following files are available from the example directory $SE_HOME/../examples:

File	Explanation
<model>.[f | c | cpp | py | gms]	model source code (check also example files in Section 15.2)
<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.edf and world_f_05x05.edf
world.post_[c | e | f | bas | adv]	post-processor input file (complete experiment) for world.edf_[c | e | f] (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>
usr_opr_<opr>.f	source code file for user-defined post-processing operator <opr>

File	Explanation
land_sea_mask[<nil> | .f]	executable and source code to derive a coarsed land-sea-mask from the file land_sea_mask.05x05
land_sea_mask.05x05	global ASCII land-sea-mask file with a resolution of 0.5° lat x 0.5° lon
read_result_file[<nil> | .f]	executable and source code for the result file import interface of ASCII and IEEE compliant result output

Tab. 15.5 *Implemented model and operator related user files for the current version*
For <opr> see Tab. 15.6 below

15.1.5 Example User-Defined Operators

The following user-defined operators are available from the example directory $SE_HOME/../examples as source code and executables <opr>.opr. All but operator matmul_c (source file usr_opr_<opr>.c) are implemented in Fortran and available as source files usr_opr_<opr>.f.

Operator name <opr>	Operator arguments	Explanation	Example
char_test	char_arg1,char_arg2, arg	character test check usr_opr_char_test.f	`char_test('arg11', 'arg22',bios)`
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`
matmul_[f | c]	arg1,arg2	matrix multiplication of 2-dimensional operands	`matmul_[f | c ] (mat1,mat2)`
simple_div	arg1,arg2	division without consideration of overflow, underflow, and division by 0.	`simple_div(-2,-4) = 0.5`

Tab. 15.6 *Available user-defined operators*

15.2 Examples for Model Interfaces

15.2.1 Example Implementation of the Generic Model world

According to Example 1.1 on 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 name	Target unit	Target 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. 15.7 *Targets of the generic model world*
*Mapping between model targets and internal model parameters is performed by the model coupling interface functions simenv_get_**

For reasons of simplification these targets (parameters) influence state variables atmo and bios by the product of two trigonometric terms value_lat and value_lon 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(.) norms value_lat and value_lon by lat and/or lon in a way, that it holds

```
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 )
```

Finally,

```
atmo(lat,lon,level,time) = value_lat(lat) * value_lon(lon) * (100*time+level-1)
bios(lat,lon,time)       = value_lat(lat) * value_lon(lon) *  100*time
```

and

```
atmo_g(time)              = avg_l('001',abs(atmo(lat,lon,1,time)))
bios_g                    = avg(abs(bios(lat,lon,time)))
```

Means avg and avg_l are calculated in a box with the extent Δlat x Δlon = 10° x 10° and (lat,lon) = (0°,0°) in the mid of the box.

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 interfaced to SimEnv. SimEnv modifications are marked in **bold**.

```
      program world_f
      c  declare SimEnv interface functions (compile with –I$SE_HOME)
      c  simenv_sts, simenv_run_int and simenv_run_char are also declared there
         include 'simenv_mod_f.inc'
      c  declare atmo without dimensions level and time and bios without time
      c  because they are computed in place and simenv_slice_f 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

         p1 = 1.
         p2 = 2.
         p3 = 3.
         p4 = 4.

         simenv_sts = simenv_ini_f()
      c  check return code for the model interface functions at least here
         if(simenv_sts.ne.0) call exit_(1)
      c  only if necessary:
         simenv_sts = simenv_get_run(simenv_run_int,simenv_run_char)
         simenv_sts = simenv_get_f('p1',p1,p1)
         simenv_sts = simenv_get_f('p2',p2,p2)
         simenv_sts = simenv_get_f('p3',p3,p3)
         simenv_sts = 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
               simenv_sts = simenv_slice_f('atmo',3,level+1,level+1)
               simenv_sts = simenv_slice_f('atmo',4,idecade+1,idecade+1)
               simenv_sts = simenv_put_f('atmo',atmo)
            enddo
            simenv_sts = simenv_slice_f('bios',3,idecade+1,idecade+1)
            simenv_sts = simenv_put_f('bios',bios)
         enddo
         ...
         simenv_sts = simenv_put_f('atmo_g',atmo_g)
      c  compute dynamics of bios_g
         ...
         simenv_sts = simenv_put_f('bios_g',bios_g)

         simenv_sts = simenv_end_f()
         end
```

Example file: world_f.f

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

15.2.3 Fortran Model with Semi-Automated Model Interface

With respect to Example 5.1 the following Fortran code **world_f_auto.f** could be used to describe the model interfaced semi-automatedly to SimEnv. SimEnv modifications are marked in **bold**.

```
      program world_f_auto
      c  declare SimEnv interface functions (compile with –I$SE_HOME)
      c  simenv_sts, simenv_run_int and simenv_run_char are also declared there
         include 'simenv_mod_f.inc'
      c  declare atmo without dimensions level and time and bios without time
      c  because they are computed in place and simenv_slice_f 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

         p1 = 1.
         p2 = 2.
         p3 = 3.
         p4 = 4.

      c  include source code sequence for the semi-automated model interface
         include 'world_f_auto_f.inc'

      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
               simenv_sts = simenv_slice_f('atmo',3,level+1,level+1)
               simenv_sts = simenv_slice_f('atmo',4,idecade+1,idecade+1)
               simenv_sts = simenv_put_f('atmo',atmo)
            enddo
            simenv_sts = simenv_slice_f('bios',3,idecade+1,idecade+1)
            simenv_sts = simenv_put_f('bios',bios)
         enddo
         ...
         simenv_sts = simenv_put_f('atmo_g',atmo_g)
      c  compute dynamics of bios_g
         ...
         simenv_sts = simenv_put_f('bios_g',bios_g)

         simenv_sts = simenv_end_f()
         end
```

Example file: world_f_auto.f

Example 15.2 *Semi-automated model interface for Fortran models - model world_f_auto.f*

15.2.4 C Model

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

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/* declare SimEnv interface functions (compile with –I$SE_HOME)
simenv_sts, simenv_run_int and simenv_run_char are also declared there */
#include "simenv_mod_c.inc"

/* declare atmo without dimensions level and time and bios without time*/
/* because they are computed in place and simenv_slice_c 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 level,idecade,level1,idecade1,idim;
   p1 = 1.;
   p2 = 2.;
   p3 = 3.;
   p4 = 4.;

   simenv_sts = simenv_ini_c();
/* check return code of model interface functions at least here */
   if(simenv_sts != 0) return 1;
/* only if necessary: */
   simenv_sts = simenv_get_run_c(&simenv_run_int,simenv_run_char);
   simenv_sts = simenv_get_c("p1",&p1,&p1);
   simenv_sts = simenv_get_c("p2",&p2,&p2);
   simenv_sts = simenv_get_c("p3",&p3,&p3);
   simenv_sts = 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;
       level1=level+1;
       simenv_sts = simenv_slice_c("atmo",&idim,&level1,&level1);
       idim=4;
       idecade1=idecade+1;
       simenv_sts = simenv_slice_c("atmo",&idim,&idecade1,&idecade1);
       simenv_sts = simenv_put_c("atmo",(char *) &atmo);
     }
     idim=3;
     idecade=idecade+1;
     simenv_sts = simenv_slice_c("bios",&idim,&idecade1,&idecade1);
     simenv_sts = simenv_put_c("bios",(char *) &bios);
   }
```

```
      simenv_sts = simenv_put_c("atmo_g",(char *) &atmo_g);

   /* compute dynamics of bios_g */
      ...
      simenv_sts = simenv_put_c("bios_g", ,(char *) &bios_g);
      simenv_sts = simenv_end_c();
      return 0;
   }
```

Example file: world_c.c

Example 15.3 *Model interface for C models – model world_c.c*

15.2.5 C++ Model

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

```
#include <stdio.h>
#include <stdlib.h>
/* declare SimEnv interface functions (compile with –I$SE_HOME)
simenv_sts, simenv_run_int and simenv_run_char are also declared there */
#include "simenv_mod_c.inc"

class World
{
/* declare atmo without dimensions level and time and bios without time*/
/* because they are computed in place and simenv_slice_c 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,level1,idecade1,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;
         level1=level1+1;
         simenv_sts = simenv_slice_c("atmo",&idim,&level,&level);
         idim=4;
         idecade1=idecade1+1;
         simenv_sts = simenv_slice_c("atmo",&idim,&idecade,&idecade);
         simenv_sts = 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;
       idecade1=idecade1+1;
       simenv_sts = simenv_slice_c("bios",&idim,&idecade1,&idecade1);
       simenv_sts = simenv_put_c("bios",(char *) &bios);
     }
/* compute dynamics of bios_g */
     ...
   }
}
```

```
main(void)
{
   float p1 = 1.;
   float p2 = 2.;
   float p3 = 3.;
   float p4 = 4.;

   simenv_sts = simenv_ini_c();
/* check return code of model interface functions at least here */
   if(simenv_sts != 0) return 1;
/* only if necessary: */
   simenv_sts = simenv_get_run_c(&simenv_run_int,simenv_run_char);

   simenv_sts = simenv_get_c("p1",&p1,&p1);
   simenv_sts = simenv_get_c("p2",&p2,&p2);
   simenv_sts = simenv_get_c("p3",&p3,&p3);
   simenv_sts = simenv_get_c("p4",&p4,&p4);

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

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

Example file: world_cpp.cpp

Example 15.4 *Model interface for C++ models – model world_cpp.cpp*

15.2.6 Python Model

With respect to Example 5.1 the following Python code **world_py.py** could be used to describe the model interfaced 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)
p1=1.
p2=2.
p3=3.
p4=4.

simenv_ini_py()
# only if necessary:
simenv_run_int = int(simenv_get_run_py())
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 15.5 *Model interface for Python models – model world_py.py*

15.2.7 Model Interface at Shell Script Level

Assume any experiment. Assume model executable world_sh to take 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:

```
#! /bin/sh

p1=1.
p2=2.
p3=3.
p4=4.

# perform always and as the first $SE_HOME/simenv_*_sh dot script
# altern. perform . $SE_WS/<model>_sh.inc for semi-autom. model interface
. $SE_HOME/simenv_ini_sh

# get current run number simenv_run_char and simenv_run_int
. $SE_HOME/simenv_get_run_sh

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

# create temporary directory run<simenv_run_char> to perform the model
# and model output transformation from native to SimEnv structure there
mkdir run$simenv_run_char
cd run$simenv_run_char

# 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
cd ..
rm -fR run$simenv_run_char

# perform always and as the last $SE_HOME/simenv_*_sh dot script
. $SE_HOME/simenv_end_sh
```

Example file: world_sh.run

Example 15.6 *Model interface at shell script level – model shell script world_sh.run*

15.2.8 Semi-Automated Model Interface at Shell Script Level

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

```
#! /bin/sh

p1=1.
p2=2.
p3=3.
p4=4.

# perform dot script world_sh_auto_sh.inc
# for semi-automated model interface at shell script level
# alternatively perform dot script $SE_HOME/simenv_ini_sh
. $SE_WS/world_sh_auto_sh.inc

# create temporary directory run<simenv_run_char> to perform the model
# and model output transformation from native to SimEnv structure there
mkdir run$simenv_run_char
cd run$simenv_run_char

# 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
cd ..
rm -fR run$simenv_run_char

# perform always and as the last $SE_HOME/simenv_*_sh dot script
. $SE_HOME/simenv_end_sh
```

Example file: world_sh_auto.run

Example 15.7 *Semi-automated model interface at shell script level – model shell script world_sh_auto.run*

15.2.9 GAMS Model

SimEnv comes with an interfaced 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 model parameters
*   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 output 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 statements are possible after the $include statement
  modstat = transport.modelstat
  $include gams_model_simenv_put.inc

* Only if sub-models sub_m1 and sub_m2 are coupled (see Example 5.5):
* $call “gams ../sub_m1.gms ll= lo=2 lf=gams_model.nlog dp=0”;
* $call “gams ../sub_m2.gms ll= lo=2 lf=gams_model.nlog dp=0”;
```

Example file: gams_model.gms

Example 15.8 *Model interface for GAMS models – model gams_model.gms*

15.3 Example Implementation for the Experiment Post-Processor User-Defined Operator matmul_[f | c]

15.3.1 Fortran Implementation

Implementation of the user-defined operator matmul_f in the file **usr_opr_matmul_f.f**:

```
      integer*4 function simenv_check_user_def_operator()
c     declare SimEnv interface functions (compile with –I$SE_HOME)
      include ‘simenv_opr_f.inc’
c     declare fields to hold extents and coordinates
      dimension iext1(9),iext2(9)
      dimension ico_nr1(9),ico_nr2(9)
      dimension ico_beg_pos1(9),ico_beg_pos2(9)
      character*20 co_name1(9),co_name2(9)

c     get dimensionality idimens, extents iext,
c     formal coordinate number ico_nr and
c     formal coordinate begin position ico_beg_pos
      idimens1=simenv_get_dim_arg_f(1,iext1)
      idimens2=simenv_get_dim_arg_f(2,iext2)
      iok=simenv_get_co_arg_f(1,ico_nr1,ico_beg_pos1,co_name1)
      iok=simenv_get_co_arg_f(2,ico_nr2,ico_beg_pos2,co_name2)
c     get check modus for coordinates
      ichk_modus=simenv_get_co_chk_modus_f()

      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_nr1(2).eq.ico_nr2(1)) then
c     coordinates identical
               if(ico_beg_pos1(2).eq.ico_beg_pos2(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=simenv_get_co_val_f(
     #                     ico_nr1(2),ico_beg_pos1(2)+j,value1)
                     iretv2=simenv_get_co_val_f(
     #                     ico_nr2(1),ico_beg_pos2(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_nr1(2)=ico_nr2(2)
       ico_beg_pos1(2)=ico_beg_pos2(2)
       iok=simenv_put_struct_res_f(0,idimens1,iext1,ico_nr1,ico_beg_pos1)
     endif

  c  return error code
     simenv_check_user_def_operator=ierror
     return
     end



     integer*4 function simenv_compute_user_def_operator(res)
  c  SimEnv operator results are always of type real*4
     real*4 res(1)
  c  declare SimEnv interface functions (compile with –I$SE_HOME)
     include 'simenv_opr_f.inc'
  c  auxiliary variables
     integer*4 iext1(9),iext2(9)
     real*8 value8

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

  c  perform matrix multiplication
     m=0
     do k=1,iext2(2)
        iarg2_offs=(k-1)*iext2(1)
        do i=1,iext1(1)
           iarg1_offs=i
  c   res(i,k) = sum(arg1(i,l) * arg2(l,k))
           value8=0.
           indi_defined=0
           do l=1,iext1(2)
              ia1=iarg1_offs+(l-1)*iext1(1)
              ia2=iarg2_offs+l
              fac1=simenv_get_arg_f(1,ia1)
              fac2=simenv_get_arg_f(2,ia2)
              if(simenv_chk_undef_f(fac1)+simenv_chk_undef_f(fac2).eq.0)
              then
```

```
                  indi_defined=1
                  value8=value8+fac1*fac2
               endif
            enddo
            m=m+1
            if(indi_defined.eq.0) then
               res(m)=simenv_put_undef_f()
            else
               res(m)=simenv_clip_undef_f(value8)
            endif
         enddo
      enddo

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

Example file: usr_opr_matmul_f.f

Example 15.9 *Experiment post-processor user-defined operator module – operator matmul_f*

15.3.2 C Implementation

Implementation of the user-defined operator matmul_c in the file **usr_opr_matmul_c.c**:

```
#include <strings.h>
#include <stdio.h>
#include “simenv_opr_c.inc”           /* compile with -I$SE_HOME */

int simenv_check_user_def_operator()
{
   int iext1[9],iext2[9];
   int ico_nr1[9],ico_nr2[9],ico_beg_pos1[9],ico_beg_pos2[9];
   char co_name1[180],co_name2[180];
   int idimens1, idimens2;
   int ichk_modus;
   int iret,iretv1,iretv2,j,iok,ierror=0;
   float value1, value2;

/* get dimensionality idimens, extents iext,
   formal coordinate number ico_nr and
   formal coordinate begin position ico_beg_pos
*/
   idimens1=simenv_get_dim_arg_c(1,iext1);
   idimens2=simenv_get_dim_arg_c(2,iext2);
   iok=simenv_get_co_arg_c(1,ico_nr1,ico_beg_pos1,co_name1);
   iok=simenv_get_co_arg_c(2,ico_nr2,ico_beg_pos2,co_name2);

   ichk_modus=simenv_get_co_chk_modus_c();

   if(idimens1!=2 || idimens2!=2)
      ierror=1;                     /* wrong dimensionalities */
   else
      if(iext1[1]!=iext2[0])
         ierror=2;                  /* wrong dimensions */
      else
         { if(ico_nr1[1]==ico_nr2[0])
              if(ico_beg_pos1[1]==ico_beg_pos2[0])
                 iret=31;
              else
                 iret=33;           /* coordinates identical*/
           else
             { iret=32;             /* differing coordinates */
               if(ichk_modus==1)
                  for (j=0;j<iext1[1];j++) /* only for weak c. check */
                     { /* get coordinate values */
                       iretv1=simenv_get_co_val_c
                              (ico_nr1[1],ico_beg_pos1[1]+j,&value1);
                       iretv2=simenv_get_co_val_c
                              (ico_nr2[0],ico_beg_pos2[0]+j,&value2);
/* iret=33: differing coordinate values */
                              if(value1 != value2)
                                 iret=33;
                     }
             }
```

```
                ierror=0;
                if(ichk_modus==2)
                   if(iret>31) ierror=3;
                else
                   if(ichk_modus==1)
                      if(iret>32) ierror=3;
              }

      if(ierror==0)
            { iext1[1]=iext2[1];
              ico_nr1[1]=ico_nr2[1];
              ico_beg_pos1[1]=ico_beg_pos2[1];
      iok=simenv_put_struct_res_c(0,idimens1,iext1,ico_nr1,
                                        ico_beg_pos1);
            }
         return ierror; /* return error code */
       }



      /* SimEnv operator results are always of type real*4 */
      int simenv_compute_user_def_operator(float *res)
      {
         int iext1[9],iext2[9];
         double value8;
         int idimens;
         int i,k,l,m,ia1,ia2;
         int iarg1_offs,iarg2_offs,indi_defined;
         float fac1,fac2;

      /* get dimensionality idimens and dimensions idim for both arguments */
         idimens=simenv_get_dim_arg_c(1,iext1);
         idimens=simenv_get_dim_arg_c(2,iext2);

      /* perform matrix multiplication */
         m=0;
         for (k=1;k<=iext2[1];k++)
            { iarg2_offs=(k-1)*iext2[0];
              for (i=1;i<=iext1[0];i++)
                 { iarg1_offs=i;
      /* res(i,k) = sum(arg1(i,l) * arg2(l,k)) */
                   value8=0.;
                   indi_defined=0;
                   for (l=1;l<=iext1[1];l++)
                      { ia1=iarg1_offs+(l-1)*iext1[0];
                        ia2=iarg2_offs+l;
                        fac1=simenv_get_arg_c(1,ia1);
                        fac2=simenv_get_arg_c(2,ia2);
                        if(simenv_chk_undef_c(fac1) +
                           simenv_chk_undef_c(fac2)==0)
                           { indi_defined=1;
                             value8=value8+fac1*fac2;
                           }
                      }
                   m=m+1;
                if(indi_defined==0)
                   res[m-1]=simenv_put_undef_c();
                else
                   res[m-1]=simenv_clip_undef_c(value8);
             }
          }
       return 0;
    }
```

Example file: usr_opr_matmul_c.c

Example 15.10 *Experiment post-processor user-defined operator module – operator matmul_c*

15.4 Example for an Experiment Post-Processor Result Import Interface

In Example 15.11 an implementation of an interface to import ASCII post-processor output from SimEnv can be found. A corresponding interface to import IEEE compliant post-processor output is documented.

```
      subroutine read_result_file_ascii(model_name,res_nmb)
      character model_name*20,res_nmb*2
      real*4, pointer, dimension(:) :: coord_values
      real*4, pointer, dimension(:) :: result_values
      integer*4 idim, iext(9)
      character result_expr*512, result_desc*128, result_unit*32
      character coord_name*20
      open(unit=1,file=trim(model_name)//‘inf’//res_nmb//‘.ascii’,
     #      form=‘formatted’,status=‘old’)
      open(unit=2,file=trim(model_name)//‘res’//res_nmb//‘.ascii’,
     #      form=‘formatted’,status=‘old’)
      iostat=0
      do while (iostat.eq.0)
         read(1,‘(a512)’,iostat=iostat) result_expr
         if(iostat.eq.0) then
            read(1,‘(a128)’,iostat=iostat1) result_desc
            read(1,‘(a32)’,iostat=iostat1) result_unit
            read(1,‘(10i8)’,iostat=iostat1) idim,(iext(i),i=1,9)
            length_result=1
            do i=1,idim
               length_result=length_result*iext(i)
               read(1,‘(a20)’,iostat=iostat1) 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)’,iostat=iostat1) (coord_values(j),
                  ibeg=iend+1                          j=ibeg,iend)
               enddo
c              further processing of coordinate values
c              ...
               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)’,iostat=iostat) (result_values(j),
               ibeg=iend+1                         j=ibeg,iend)
            enddo
c           further processing of result values
c           ...
            deallocate(result_values)
         endif
      enddo
      close(unit=1)
      close(unit=2)
      return
      end
```

Example file: read_result_file.f (together with subroutine read_result_file_ieee)

Example 15.11 *ASCII compliant experiment post-processor result import interface*

15.5 List of Experiment Post-Processor Built-In Operators and Operator Arguments

15.5.1 Experiment Post-Processor Built-In Operators (in Thematic Order)

arg	general numerical argument
int_arg	integer constant argument ≥ 0
real_arg	real (float) constant argument
char_arg	character argument

Name	Meaning	See
Elemental operators		**Tab. 8.3 on page 70**
arg1 + arg2	addition	
arg1 - arg2	subtraction	
arg1 * arg2	multiplication	
arg1 / arg2	division	
arg1 ** arg2	exponentiation	
+ arg	identity	
- arg	negation	
(arg)	parentheses	
Basic operators		**Tab. 8.4 on page 71**
abs(arg)	absolute value	
dim(arg1,arg2)	positive difference	
exp(arg)	exponential function	
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		**Tab. 8.4 on page 71**
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	
Advanced operators		**Tab. 8.8 on page 77**
classify(int_arg1, real_arg2,real_arg3,arg4)	classification of arg4 into int_arg1 classes	
clip(char_arg1,arg2)	clip arg2 according to char_arg1	
cumul(char_arg1,arg2)	cumulates arg2 according to char_arg1	

Name	Meaning See
flip(char_arg1,arg2)	flip arg2 according to char_arg1
get_data(char_arg1, char_arg2,char_arg3,arg4)	get data from an external file
get_experiment(char_arg1, char_arg2,char_arg3,arg4)	include an other experiment
get_table_fct(char_arg1,arg2)	table function with linear interpolation of table char_arg1 for position arg2
if(char_arg1,arg2,arg3,arg4)	general purpose conditional if-construct
mask(char_arg1,arg2,arg3)	mask elements of argument arg21
matmul(arg1,arg2)	matrix multiplication
move_avg(char_arg1, char_arg2,int_arg3,arg4)	moving average of running length int_arg3 for arg4
nr_of_runs()	number of single runs of the current experiment
rank(char_arg1,arg2)	rank of arg2 according to char_arg1
regrid(char_arg1,arg2)	assign new coordinates to arg2
run(char_arg1,arg2)	values of arg2 for a single run selected by char_arg1
transpose(char_arg1,arg2)	transpose arg2 according to char_arg1
undef( )	undefined element
Aggregation and moment operators for arguments	**Tab. 8.5 on page 73**
avg(arg)	argument arithmetic mean of values
avgg(arg)	argument geometric mean of values
avgh(arg)	argument harmonic mean of values
avgw(arg1,arg2)	argument weighted mean of values
count(char_arg1,arg2)	count number of values according to char_arg1
hgr(char_arg1,int_arg2, real_arg3,real_arg4, arg5)	argument histogram of values
max(arg)	argument maximum of values
maxprop(arg)	index of the element where the maximum is reached the first time
min(arg)	argument minimum of values
minprop(arg)	index of the element where the minimum is reached the first time
sum(arg)	argument sum of values
var(arg)	argument variance of values
Multiple aggregation and moment operators for arguments	**Tab. 8.6 on page 73**
max_n(arg1 ,..., argn)	maximum per element
maxprop_n(arg1 ,..., argn)	argument position (1 ... n) where the maximum is reached the first time
min_n(arg1 ,..., argn)	minimum per element
minprop_n(arg1 ,..., argn)	argument position (1 ... n) where the minimum is reached the first time
Dimension related aggregation and moment operators for arguments	**Tab. 8.7 on page 74**
avg_l(char_arg1,arg2)	dimension related argument arithmetic means of values of arg2
avgg_l(char_arg1,arg2)	dimension related argument geometric means of values of arg2
avgh_l(char_arg1,arg2)	dimension related argument harmonic means of values of arg2
avgw_l(char_arg1,arg2,arg3)	dimension related argument weighted means of values of arg2
count_l(char_arg1,char_arg2, arg3)	dimension related count numbers of values of arg3
hgr_l(char_arg1,char_arg2, int_arg3,real_arg4, real_arg5,arg6)	dimension related argument histograms of values of arg6
max_l(char_arg1,arg2)	dimension related argument maxima of values of arg2
maxprop_l(char_arg1,arg2)	dimension related argument position (1 ... n) where the maximum of arg2 is reached the first time
min_l(char_arg1,arg2)	dimension related argument minima of values of arg2

Name	Meaning	See
minprop_l(char_arg1,arg2)	dimension related argument position (1 ... n) where the minimum of arg2 is reached the first time	
sum_l(char_arg1,arg2)	dimension related argument sums of values of arg2	
var_l(char_arg1,arg2)	dimension related argument variances of values of arg2	
Multi-run operators (behavioural analysis)		**Tab. 8.10 on page 84**
behav(char_arg1,arg2)	general purpose operator for navigating and aggregating arg2 in the experiment space	
Multi-run operators (Monte Carlo analysis and optimization)		**Tab. 8.12 on page 88** **Tab. 8.9 on page 83**
avg_e(arg)	run ensemble mean	
avgg_e(arg)	run ensemble geometric mean	
avgh_e(arg)	run ensemble harmonic mean	
avgw_e(arg1,arg2)	run ensemble weighted mean	
cnf(real_arg1,arg2)	positive distance of confidence line from mean avg_e(arg2)	
cor(arg1,arg2)	correlation coefficient between arg1 and arg2	
count_e(char_arg1,arg2)	run ensemble count number of values	
cov(arg1,arg2)	covariance between arg1 and arg2	
ens(arg)	whole Monte Carlo run ensemble	
hgr_e(char_arg1,int_arg2, real_arg3,real_arg4,arg5)	heuristic probability density function	
krt(arg)	kurtosis (4th moment)	
max_e(arg)	run ensemble maximum	
maxprop_e(arg)	run number where the maximum is reached the first time	
med(arg)	median	
min_e(arg)	run ensemble minimum	
minprop_e(arg)	run number where the minimum is reached the first time	
qnt(real_arg1,arg2)	quantile of arg2	
reg(arg1,arg2)	linear regression coefficient to forecast arg2 from arg1	
rng(arg)	range = max_e(arg) - min_e(arg)	
skw(arg)	skewness (3rd moment)	
stat_full(real_arg1,real_arg2, real_arg3,real_arg4,arg5)	full basic statistical measures	
stat_red(real_arg1,real_arg2, arg3)	reduced basic statistical measures	
sum_e(arg)	run ensemble sum	
var_e(arg)	run ensemble variance	
Multi-run operators (local sensitivity analysis)		**Tab. 8.13 on page 90**
lin_abs(char_arg1,arg2)	absolute linearity measure	
lin_rel(char_arg1,arg2)	relative linearity measure	
sens_abs(char_arg1,arg2)	absolute sensitivity measure	
sens_rel(char_arg1,arg2)	relative sensitivity measure	
sym_abs(char_arg1,arg2)	absolute symmetry measure	
sym_rel(char_arg1,arg2)	relative symmetry measure	

Tab. 15.8 *Experiment post-processor built-in operators (in thematic order)*

15.5.2 Experiment Post-Processor Built-In Operators (in Alphabetic Order)

arg	general numerical argument
int_arg	integer constant argument ≥ 0
real_arg	real (float) constant argument
char_arg	character argument

Name	Meaning	Type	See	At page
arg1 + arg2	addition	elemental	Tab. 8.3	70
arg1 - arg2	subtraction	elemental	Tab. 8.3	70
arg1 * arg2	multiplication	elemental	Tab. 8.3	70
arg1 / arg2	division	elemental	Tab. 8.3	70
arg1 ** arg2	exponentiation	elemental	Tab. 8.3	70
+ arg	identity	elemental	Tab. 8.3	70
- arg	negation	elemental	Tab. 8.3	70
(arg)	parentheses	elemental	Tab. 8.3	70
abs(arg)	absolute value	basic	Tab. 8.4	71
acos(arg)	arc cosine	trigonom.	Tab. 8.4	71
acot(arg)	arc cotangent	trigonom.	Tab. 8.4	71
asin(arg)	arc sine	trigonom.	Tab. 8.4	71
atan(arg)	arc tangent	trigonom.	Tab. 8.4	71
avg(arg)	argument arithmetic mean of values	aggr./mom.	Tab. 8.5	73
avg_e(arg)	run ensemble mean	Monte C.	Tab. 8.9	83
avg_l(char_arg1,arg2)	dimension related argument arithmetic means of values of arg2	aggr./mom.	Tab. 8.7	74
avgg(arg)	argument geometric mean of values	aggr./mom.	Tab. 8.5	73
avgg_e(arg)	run ensemble geometric mean	Monte C.	Tab. 8.9	
avgg_l(char_arg1,arg2)	dimension related argument geometric means of values of arg2	aggr./mom.	Tab. 8.7	74
avgh(arg)	argument harmonic mean of values	aggr./mom.	Tab. 8.5	73
avgh_e(arg)	run ensemble harmonic mean	Monte C.	Tab. 8.9	
avgh_l(char_arg1,arg2)	dimension related argument harmonic means of values of arg2	aggr./mom.	Tab. 8.7	74
avgw(arg1,arg2)	argument weighted mean of values	aggr./mom.	Tab. 8.5	73
avgw_e(arg1,arg2)	run ensemble weighted mean	Monte C.	Tab. 8.9	
avgw_l(char_arg1,arg2, arg3)	dimension related argument weighted means of values of arg3	aggr./mom.	Tab. 8.7	74
behav(char_arg1,arg2)	general purpose operator for navigating and aggregating of arg2 in the experiment space	behav.	Tab. 8.10	84
classify(int_arg1,real_arg2, real_arg3,arg4)	classification of arg4 into int_arg1 classes	advanced	Tab. 8.8	77
clip(char_arg1,arg2)	clip arg2 according to char_arg1	advanced	Tab. 8.8	77
cnf(real_arg1,arg2)	positive distance of confidence line from mean avg_e(arg2)	Monte C.	Tab. 8.12	83
cor(arg1,arg2)	correlation coefficient between arg1 and arg2	Monte C.	Tab. 8.12	88
cos(arg)	cosine	trigonom.	Tab. 8.4	71
cosh(arg)	hyperbolic cosine	trigonom.	Tab. 8.4	71
cot(arg)	cotangent	trigonom.	Tab. 8.4	71
coth(arg)	hyperbolic cotangent	trigonom.	Tab. 8.4	71
count(char_arg1,arg2)	count number of values	aggr./mom.	Tab. 8.5	73
count_e(char_arg1,arg2)	run ensemble count	Monte C.	Tab. 8.9	83

Name	Meaning	Type	See	At page
count_l(char_arg1, char_arg2,arg3)	dimension related count numbers of values of arg3	aggr./mom.	Tab. 8.7	74
cov(arg1,arg2)	covariance between arg1 and arg2	Monte C.	Tab. 8.12	88
cumul(char_arg1,arg2)	cumulates arg2 according to char_arg1	advanced	Tab. 8.8	77
dim(arg1,arg2)	positive difference	basic		71
ens(arg)	whole Monte Carlo run ensemble	Monte C.	Tab. 8.12	88
exp(arg)	exponential function	basic	Tab. 8.4	71
flip(char_arg1,arg2)	flip arg2 according to char_arg1	advanced	Tab. 8.8	77
get_data(char_arg1, char_arg2,char_arg3,arg4)	get data from an external file	advanced	Tab. 8.8	77
get_experiment(char_arg1, char_arg2,char_arg3,arg4)	include an other experiment	advanced	Tab. 8.8	77
get_table_fct(char_arg1, arg2)	table function with linear interpolation of table char_arg1 for position arg2	advanced	Tab. 8.8	77
hgr(char_arg1,int_arg2, real_arg3,real_arg4,arg5)	argument histogram of values	aggr./mom.	Tab. 8.5	73
hgr_e(char_arg1,int_arg2, real_arg3,real_arg4,arg5)	heuristic probability density function	Monte C.	Tab. 8.9	83
hgr_l(char_arg1,char_arg2, int_arg3,real_arg4, real_arg5,arg6)	dimension related argument histograms of values of arg6	aggr./mom.	Tab. 8.7	74
if(char_arg1,arg2,arg3,arg4)	general purpose conditional if-construct	advanced	Tab. 8.8	77
int(arg)	truncation value	basic	Tab. 8.4	71
krt(arg)	kurtosis (4th moment)	Monte C.	Tab. 8.12	88
lin_abs(char_arg1,arg2)	absolute linearity measure	sensitivity	Tab. 8.13	90
lin_rel(char_arg1,arg2)	relative linearity measure	sensitivity	Tab. 8.13	90
log(arg)	natural logarithm	basic	Tab. 8.4	71
log10(arg)	decade logarithm	basic	Tab. 8.4	71
mask(char_arg1,arg2,arg3)	mask elements of argument arg2	advanced	Tab. 8.8	77
matmul(arg1,arg2)	matrix multiplication	advanced	Tab. 8.8	77
max(arg)	argument maximum of values	aggr./mom.	Tab. 8.5	73
max_e(arg)	run ensemble maximum	Monte C.	Tab. 8.9	83
max_l(char_arg1,arg2)	dimension related argument maxima of values of arg2	aggr./mom.	Tab. 8.7	74
max_n(arg1 ,..., argn)	maximum per element	aggr./mom.	Tab. 8.5	73
maxprop(arg)	index of the element where the maximum is reached the first time	aggr./mom.	Tab. 8.5	73
maxprop_e(arg)	run number where the maximum is reached the first time	Monte C.	Tab. 8.12	83
maxprop_l(char_arg1,arg2)	dimension related argument position (1 ... n) where the maximum is reached the first time of arg2	aggr./mom.	Tab. 8.7	74
maxprop_n(arg1 ,..., argn)	argument position (1 ... n) where the maxi-mum is reached the first time	aggr./mom.	Tab. 8.5	73
med(arg)	median	Monte C.	Tab. 8.12	88
min(arg)	argument minimum of values	aggr./mom.	Tab. 8.5	73
min_e(arg)	run ensemble minimum	Monte C.	Tab. 8.9	
min_l(char_arg1,arg2)	dimension related argument minima of values of arg2	aggr./mom.	Tab. 8.7	74
min_n(arg1 ,..., argn)	minimum per element	aggr./mom.	Tab. 8.5	73
minprop(arg)	index of the element where the minimum is reached the first time	aggr./mom.	Tab. 8.5	73

Name	Meaning	Type	See	At page
minprop_e(arg)	run number where the minimum is reached the first time	Monte C.	Tab. 8.9	83
minprop_l(char_arg1,arg2)	dimension related argument position (1 ... n) where the minimum is reached the first time of arg2	aggr./mom.	Tab. 8.7	74
minprop_n(arg1 ,..., argn)	argument position (1 ... n) where the minimum is reached the first time	aggr./mom.	Tab. 8.5	73
mod(arg1,arg2)	remainder	basic	Tab. 8.4	71
move_avg(char_arg1, char_arg2,int_arg3,arg4)	moving average of running length int_arg3 for arg4	advanced	Tab. 8.8	77
nint(arg)	round value	basic	Tab. 8.4	71
nr_of_runs()	number of single runs of the current experiment	advanced	Tab. 8.8	77
qnt(real_arg1,arg2)	quantile of arg2	Monte C.	Tab. 8.12	88
rank(char_arg1,arg2)	rank of arg2 according to char_arg1	advanced	Tab. 8.8	77
reg(arg1,arg2)	linear regression coefficient to forecast arg2 from arg1	Monte C.	Tab. 8.12	88
regrid(char_arg1,arg2)	assign new coordinates to arg2	advanced	Tab. 8.8	77
rng(arg)	range = max_e(arg) - min_e(arg)	Monte C.	Tab. 8.12	88
run(char_arg1,arg2)	values of arg2 for a single run selected by char_arg1	advanced	Tab. 8.8	77
sens_abs(char_arg1,arg2)	absolute sensitivity measure	sensitivity	Tab. 8.13	90
sens_rel(char_arg1,arg2)	relative sensitivity measure	sensitivity	Tab. 8.13	90
sign(arg)	sign of value	basic	Tab. 8.4	71
sin(arg)	sine	basic	Tab. 8.4	71
sinh(arg)	hyperbolic sine	trigonom.	Tab. 8.4	71
skw(arg)	skewness (3rd moment)	Monte C.	Tab. 8.12	88
sqrt(arg)	square root	trigonom.	Tab. 8.4	71
stat_full(real_arg1, real_arg2,real_arg3, real_arg4,arg5)	full basic statistical measures	Monte C.	Tab. 8.12	88
stat_red(real_arg1, real_arg2,arg3)	reduced basic statistical measures	Monte C.	Tab. 8.12	88
sum(arg)	argument sum of values	aggr./mom.	Tab. 8.5	73
sum_e(arg)	run ensemble sum	Monte C.	Tab. 8.9	83
sum_l(char_arg1,arg2)	dimension related argument sums of values of arg2	aggr./mom.	Tab. 8.7	74
sym_abs(char_arg1,arg2)	absolute symmetry measure	sensitivity	Tab. 8.13	90
sym_rel(char_arg1,arg2)	relative symmetry measure	sensitivity	Tab. 8.13	90
tan(arg)	tangent	trigonom.	Tab. 8.4	71
tanh(arg)	hyperbolic tangent	trigonom.	Tab. 8.4	71
transpose(char_arg1,arg2)	transpose arg2 according to char_arg1	advanced	Tab. 8.8	77
undef( )	undefined element	advanced	Tab. 8.8	77
var(arg)	argument variance of values	aggr./mom.	Tab. 8.5	73
var_e(arg)	run ensemble variance	Monte C.	Tab. 8.9	83
var_l(char_arg1,arg2)	dimension related argument variances of values of arg2	aggr./mom.	Tab. 8.7	74

Tab. 15.9 *Experiment post-processor built-in operators (in alphabetical order)*

15.5.3 Character Arguments of Experiment Post-Processor Built-In Operators

Tab. 15.10 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)	Re-mark
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	(**)
flip	1	sequence of digits 0 and 1	(**)
get_data	1	ascii	
get_data	2	{<directory>/}<file_name>	
get_data	3	{<directory>/}<file_name>	(*)
get_experiment	1	<directory>	
get_experiment	2	<model>	
get_experiment	3	{<directory>/}<file_name>	(*)
get_table_fct	1	{<directory>/}<file_name>	
hgr	1	[bin_no | bin_mid]	
hgr_e	1	[bin_no | bin_mid]	
hgr_l	1	sequence of digits 0 and 1	(**)
hgr_l	2	[bin_no | bin_mid]	
if	1	[< | <= | > | >= | = | != | def | undef]	
lin_abs	1	(not pre-defined, case insensitive)	(*)
lin_rel	1	(not pre-defined, case insensitive)	(*)
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	(**)
move_avg	1	sequence of digits 1 to 9	(**)
move_avg	2	[lin | exp]	
rank	1	[tie_plain | tie_min | tie_avg]	
regrid	1	ascii	
run	1	[run number | not pre-defined]	
sens_abs	1	(not pre-defined, case insensitive)	(*)
sens_rel	1	(not pre-defined, case insensitive)	(*)
sum_l	1	sequence of digits 0 and 1	(**)
sym_abs	1	(not pre-defined, case insensitive)	(*)
sym_rel	1	(not pre-defined, case insensitive)	(*)
transpose	1	sequence of digits 1 to 9	(**)
var_l	1	sequence of digits 0 and 1	(**)

Tab. 15.10 *Character arguments of experiment post-processor built-in operators*

(*) Character argument can be empty

(**) The length of the character argument from a sequence of digits corresponds with the dimensionality of the non-character and non-constant argument under investigation.

15.5.4 Constant Arguments of Experiment Post-Processor Built-In Operators

Tab. 15.11 summarises for built-in operators constant argument values.

Operator	Argument number	Argument type	Argument value restriction
classify	1	int_arg	[0 | ≥ 2]
classify	2	real_arg	[arg2 = arg3 = 0. | arg2 < arg3]
classify	3	real_arg	
cnf	1	real_arg	[0.001 | 0.01 | 0.05 | 0.1]
hgr	2	int_arg	[0 | ≥ 4]
hgr	3	real_arg	[arg3 = arg4 = 0. | arg3 < arg4]
hgr	4	real_arg	
hgr_e	2	int_arg	[0 | ≥ 4]
hgr_e	3	real_arg	[arg3 = arg4 = 0. | arg3 < arg4]
hgr_e	4	real_arg	
hgr_l	3	int_arg	[0 | ≥ 4]
hgr_l	4	real_arg	[arg4 = arg5 = 0. | arg4 < arg5]
hgr_l	5	real_arg	
move_avg	3	int_arg	[0 | ≥ 3]
stat_full	1	real_arg	[0.001 | 0.01 | 0.05 | 0.1] arg1 < arg2
stat_full	2	real_arg	
stat_full	3	real_arg	0. ≤ arg3 < arg 4 ≤ 100.
stat_full	4	real_arg	
stat_red	1	real_arg	[0.001 | 0.01 | 0.05 | 0.1] arg1 < arg2
stat_red	2	real_arg	

Tab. 15.11 *Constant arguments of experiment post-processor built-in operators*

15.6 Additionally Used Symbols for the Model and Operator Interface

Tab. 15.12 lists these symbols (subroutine, function and common block names) that are linked in addition to the SimEnv model interface functions in Tab. 5.5 from the object libraries $SE_HOME/libsimenv.a and /usr/local/lib/libnetcdf.a to a Fortran and C/C++ user model when interfacing it to SimEnv. Additionally, the logical unit numbers (luns) 998 and 999 are used.

Used symbols
csimenv_<string>
isimenv_<string>
jsimenv_<string>
<string>_nc_<string>
nc<string>
nf_<string>
c2f_dimids
cdf_routine_name
f2c_coords
f2c_counts
f2c_dimids
f2c_maps
f2c_strides
read_numrecs
write_numrecs

Tab. 15.12 *Additionally used symbols for the model interface*

Tab. 15.13 lists these symbols (subroutine, function and common block names) that are linked in addition to the SimEnv operator interface functions in Tab. 8.16 and Tab. 8.17 from the object library $SE_HOME/libsimenv.a to a user-defined experiment post-processing operator.

Used symbols
csimenv_<string>
isimenv_<string>
jsimenv_<string>

Tab. 15.13 *Additionally used symbols for the operator interface*

15.7 Glossary

The glossary defines and/or explains terms in that sense they are used in this User 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 and on request in post-processing output 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: Each → dimension of a → variable and each → operand of an → operator in a → result 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 output variable with a dimensionality greater than 0 resides at a assigned (multi-dimensional) → grid. Assignments for variables is done in the model output description → user-defined file.

Coupling: → model interface

Data type: The type of a → variable as declared in the → model and the corresponding model output 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 → the model interface at the language level also in the model code.

Dimension: → dimensionality

Dimensionality dim: The number of dimensions of a model → variable or of an → operator result in → experiment post-processing. In the model output 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.

Dot script: A sequence of → Unix / → Linux operating system commands stored in an → ASCII file. The sequence of operating system commands is directly interpreted and executed by a command line interpreter, the so-called shell. Contrary to → shell scripts a child shell is not spawned. A dot script is preceded by a dot and a space when calling it.

Environment variable: At → Unix / → Linux 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 use of environment variables in directory strings <direct> is allowed.

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 post-processing: The work step of processing model output data from the whole run ensemble after performing a simulation → experiment. SimEnv post-processing enables navigation in the → target space that is sampled by an experiment as well as construction of additional output functions by declaration and computation of → results.

Experiment post-processing operator: → operator

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, → local sensitivity analysis, and → optimization.

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

Expression: → result expression

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

```
ipointer = 0
do $i_{dim}$ = 1 , $ext_{dim}$
  do $i_{dim-1}$ = 1 , $ext_{dim-1}$
    ...
     do $i_2$ = 1 , $ext_2$
        do $i_1$ = 1 , $ext_1$
           ipointer = ipointer + 1
           vector(ipointer) = field($i_1$ , $i_2$ ,..., $i_{dim-1}$ , $i_{dim}$)
        enddo
     enddo
    ...
  enddo
enddo
```

For a two-dimensional matrix this storage model corresponds to a column by column storage of the matrix to the vector, starting with the first column and for each column starting with the first row.

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 → experiment 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 number 754 for binary storage of numbers in floating point representation.

Linux: Linux is a free → Unix-type operating system (http.//www.linux.org) originally created by Linus Torvalds with the assistance of developers around the world. SimEnv runs under the SUSE-Linux implementation (http://www.suse.com) for Intel-based hardware and compatibles.

Load Leveler: The load leveler LoadL is a job management system that handles compute resources at IBM's p655 cluster at PIK.

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. During → experiment post-processing sensitivity, linearity, and symmetry measures can be determined.

Macro: An abbreviation for a unique → result expression to apply during → experiment post-processing. Macros can be embedded into result expressions and are plugged into the expression during its evaluation and computation. Macros are described in the macro description → user-defined file.

Model: A model is a deterministic or stochastic algorithm, implemented 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 coupled in the process of → interfacing the model to SimEnv.

Model coupling: → model interface

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

Model output 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 and → experiment 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 → experiment 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 the values of the operator result during → experiment 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.

Optimization: → Experiment type to minimize a cost function (objective function) over a bounded → target space. In SimEnv a simulated annealing strategy (check Section 4.5 for explanation) is used to optimize the cost function that is formed from model → variables. Often the cost function represents a distance between model output and reference data to find an optimal point in the target space that fits best the model behaviour with respect to the reference data.

Parallel Operating Environment: → POE

POE: The **P**arallel **O**perating **E**nvironment POE on IBM's p655 cluster at PIK supplies services to allocate nodes, assign jobs to nodes and launch jobs.

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 a portable, interpreted, interactive, object-oriented programming language. It incorporates modules, exceptions, dynamic typing, very high level dynamic data types, and classes.

Result: In SimEnv → experiment post-processing a result (synonym: output function) is derived from model output of the → experiment and from reference data. A result is specified by a result expression, optionally prefixed by a result description and a result unit string.

Result expression: A chain of → operators from built-in or user-defined operators applied to model output → variables and/or reference data. A result expression is a part of an → experiment post-processing → result.

Shell script: A sequence of → Unix / → Linux operating system commands stored in an → ASCII file. A shell script is interpreted and executed by a command line interpreter, the so-called shell. Contrary to → dot scripts a child shell is spawned when calling a shell script that inherits the → environment variables of the father (calling) shell. After returning to the father shell it does not transfer the environment variables and other variables of the child shell to the father shell. SimEnv demands the Bourne shell sh.

Simulation: Performing → experiments with → models

Target: Element of the input set of a → model. Targets are manipulated numerically during an → experiment. Targets can be addressed in → experiment post-processing and they have there a → dimensionality of 0.

Target adjustment: → adjustment

Unix: A computer operating system (http://www.unix.org), originally developed at AT&T/USL. SimEnv runs under the AIX Unix implementation for RS6000 hardware and compatibles from IBM.

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 output format. Variables are defined in the model output description → user file and they are output from the model to SimEnv data structures. Each variable has a unique → data type, a → dimensionality, → extents and an assigned → grid. Normally, a variable consists of a series of values, forming a field.

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

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