arXiv:0708.4233 (also based on LC-TOOL-2001-039 (revised))

WHIZARD 2.0

A generic Monte-Carlo integration and event generation package for multi-particle processes

MANUAL [1]

Wolfgang Kilian,[2] Thorsten Ohl,[3] Jürgen Reuter[4]

Deutsches Elektronen-Synchrotron DESY, D–22603 Hamburg, Germany

[1]This work is supported by Helmholtz-Alliance "Physics at the Terascale". In former stages this work has also been supported by the Helmholtz-Gemeinschaft VH–NG–005

[2]e-mail: kilian@hep.physik.uni-siegen.de

[3]e-mail: ohl@physik.uni-wuerzburg.de

[4]e-mail: reuter@physik.uni-freiburg.de

ABSTRACT

WHIZARD is a program system designed for the efficient calculation of multi-particle scattering cross sections and simulated event samples. The events can be written to file in various formats (including HepMC, LHEF, STDHEP, and ASCII) or analyzed directly on the parton level using a built-in LaTeX-compatible graphics package.

Tree-level matrix elements are generated automatically for arbitrary partonic processes by calling the built-in matrix-element generator O'Mega. Various models beyond the SM are implemented, in particular, the MSSM is supported with an interface to the SUSY Les Houches Accord input format. Matrix elements obtained by alternative methods (e.g., including loop corrections) may be interfaced as well. Using an adaptive multi-channel method for phase space integration, the program is able to calculate numerically stable signal and background cross sections and generate unweighted event samples with reasonable efficiency for processes with up to eight and more final-state particles. Polarization is treated exactly for both the initial and final states. Quark or lepton flavors can be summed over automatically where needed.

For hadron collider physics, an interface to the LHAPDF library is provided. Also, the standard PDF library (PDFLIB) can be linked. **Is this still be true?**

For Linear Collider physics, beamstrahlung (CIRCE), Compton and ISR spectra are included for electrons and photons. Alternatively, beam-crossing events can be read directly from file.

For showering, fragmenting and hadronizing the final state, a PYTHIA and HERWIGinterface are provided which follow the Les Houches Accord. WHIZARD does come with its own shower.

The WHIZARD distribution is available at

http://whizard.event-generator.org or via
http://projects.hepforge.org/whizard

Contents

Chapter 1

Introduction

WHIZARD is a modern Monte-Carlo

Chapter 2

Installation

2.1 Prerequisites and Installation

The concept of the WHIZARD installation has been changed from version 1 to version 2. Now WHIZARD is centrally installed on a computer, e.g. in the /usr/local, and then the user has a working space which is completely separated from the WHIZARD installation directory. The WHIZARD tarball can be downloaded either from the WHIZARD webpage, http://whizard.event-generator.org, or the corresponding HepForge webpage, http://projects.hepforge.org/whizard. On the WHIZARD webpage, one can either download the tarball of the most recent version (or older versions), or one can check out the latest version from the subversion (svn) repository. The latter is only recommended for developers and users willing to accept that maybe not all newly installed features are already working. The check-out from the svn repository is done with the following command:

```
svn checkout http://svn.hepforge.org/whizard/trunk/ SomeLocalDir
```

Note again, that the subversion contains the latest developer version. In order to be able, to compile this, one has to first generate the configure script out of the file configure.ac by running autoreconf (NOT autoconf) which is part of the autoconf/automake (http://www.gnu.org/software/autoconf/ and http://www.gnu.org/software/automake) package. Furthermore, the development version also needs the noweb tools to be installed on the system in order to extract the source codes and documentation from several so called .nw files. The noweb package can be downloaded and installed from http://www.cs.tufts.edu/~nr/noweb/.

The general prerequisites for the installation (i.e. also from the tarball, not only from the svn) are standard tools for software development like make etc., and two different compilers, a FORTRAN2003 for the WHIZARD core and its corresponding libraries as well as an O'Caml compiler for the O'Mega matrix element generator.

Unpack the tarball, go to the WHIZARD directory, create a new directory and go to it. In that directory, perform a ../configureFC=⟨yourFORTRANcompiler⟩--prefix=/usr/local. Note that this is because the source and compile directories should be different to avoid any problems during compilation and installation. ../configure--help shows you the options for the configure process you have. The FC environment variable allows you to specify your

FORTRAN compiler of choice. Note that WHIZARD 2 has been written in FORTRAN2003 in a fully object-oriented way. We highly recommend usage of the standard gfortran compiler from version 4.5.0 on. You can access the help menu of configure by ../configure --help. ./configure -V shows you the actual version of your downloaded WHIZARD distribution. The possible environment variables are:

```
CC          C compiler command
CFLAGS      C compiler flags
LDFLAGS     linker flags, e.g. -L<lib dir> if you have libraries in a
            nonstandard directory <lib dir>
LIBS        libraries to pass to the linker, e.g. -l<library>
CPPFLAGS    C/C++/Objective C preprocessor flags, e.g. -I<include dir> if
            you have headers in a nonstandard directory <include dir>
CPP         C preprocessor
FC          Fortran compiler command
FCFLAGS     Fortran compiler flags
CXX         C++ compiler command
CXXFLAGS    C++ compiler flags
CXXCPP      C++ preprocessor
```

For most of these there is no need to be set during installation.

The configure process checks for the build and host system type; only if this is not detected automatically, the user would have to specify this by himself. After that system-dependent files are searched for, LaTeX and Acroread for documentation and plots, the FORTRAN compiler is checked, and finally the O'Caml compiler. The next step is the checks for external programs like LHAPDF and HepMC. Finally, all the Makefiles are being built.

The compilation is done by invoking make and finally make install. You could also do a make check in order to test whether the compilation has produced sane files on your system. This is highly recommended.

Be aware that there be problems for the installation if the install path or a user's home directory is part of an AFS file system. Several times problems were encountered connected with conflicts with permissions inside the OS permission environment variables and the AFS permission flags which triggered errors during the make install procedure.

It is possible to compile WHIZARD without the O'Caml parts of O'Mega, namely by using the --disable-omega option of the configure. This will result in a built of WHIZARD with the O'Mega Fortran library, but without the binaries for the matrix element generation. All selftests (cf. 2.1.1) requiring O'Mega matrix elements are thereby switched off. Note that you can install such a built (e.g. on a batch system without O'Caml installation), but the try to build a distribution (all make distxxx targets) will fail.

2.1.1 WHIZARD self tests/checks

WHIZARD has a number of self-consistency checks and test which assure that most of its features are running in the intended way. The standard procedure to invoke these self tests

is to perform a `make check` from the `build` directory. If `src` and `build` directories are the same, all relevant files for these self-tests reside in the `test` subdirectory of the main WHIZARD directory. In that case, one could in principle just call the scripts individually from the command line. Note, that if `src` and `build` directory are different as recommended, then the input files will have been installed in `prefix/share/whizard/test`, while the corresponding test shell scripts remain in the `srcdir/test` directory. As the main shell script `run_whizard_sh` has been built in the `build` directory, one now has to copy the files over by and set the correct paths by hand, if one wishes to run the test scripts individually. `make check` still correctly performs all WHIZARD self-consistency tests.

There are additional, quite extensiv numerical tests for validation and backwards compatibility checks for SM and MSSM processes. As a standard, these extended self tests are not invoked. However, they can be enabled by setting the configure option `--enable-extnum-checks`. On the other hand, the standard self-consistency checks can be completely disabled with the option `--disable-default-checks`.

2.2 Setting up a user work space

When `WHIZARD` is installed on a system it can be used by any user in a multi-user environment.

Chapter 3

How to use WHIZARD

3.1 Getting Started

WHIZARD can run as a stand-alone program. You (the user) can steer WHIZARD either interactively or by a script file. We will first describe the latter method, since it will be the most common way to interact with the WHIZARD system.

3.1.1 Hello World

The script is written in SINDARIN. This is a DSL – a domain-specific scripting language that is designed for the single purpose of steering WHIZARD.

Previous versions of the program, similar to most high-energy physics programs, relied on a bunch of input files that the user had to provide in some obfuscated format. This approach is sufficient for straightforward applications. However, once you get experienced with a program, you start thinking about uses that the program's authors did not foresee. In case of a Monte Carlo package, typical abuses are parameter scans, complex patterns of cuts and reweighting factors, or data analysis without recourse to external packages. This requires more flexibility.

Instead of transferring control over data input to some generic scripting language like PERL or PYTHON (or even C++), which come with their own peculiarities and learning curves, we decided to unify data input and scripting in a dedicated steering language that is particularly adapted to the needs of Monte-Carlo integration, simulation, and simple analysis of the results. Thus we discovered what everybody knew anyway: that W(h)izards communicate in SINDARIN, Scripting INtegration, Data Analysis, Results display and INterfaces.

Now since SINDARIN *is* a programming language, we honor the old tradition of starting with the famous Hello World program. In SINDARIN this reads

```
echo ("Hello World!")
```

Open your favorite editor, type this text, and save it into a file named `hello.sin`.

Now we assume that you – or your kind system administrator – has installed WHIZARD in your executable path. Then you should open a command shell and execute

```
> whizard -r hello.sin
```

and if everything works well, you get the output

```
| Writing log to 'whizard.log'
|================================================================================|
|                                WHIZARD 2.0.0_rc1
|================================================================================|
| Initializing process library 'processes'
| Reading model file 'SM.mdl'
| Using model: SM
| Reading commands from file 'hello.sin'
Hello World!
| WHIZARD run finished.
|================================================================================|
```

3.1.2 A Simple Calculation

You may object that WHIZARD is not exactly designed for printing out plain text. So let us demonstrate a more useful example.

Looking at the Hello World output, we first observe that the program writes a log file named (by default) `whizard.log`. This file receives all screen output, except for the output of external programs that are called by WHIZARD. You don't have to cache WHIZARD's screen output yourself.

After the welcome banner, WHIZARD tells you that it initializes a *process library*, and it reads a physics *model*. The process library is initially empty. It is ready for receiving definitions of elementary high-energy physics processes (scattering or decay) that you provide. The processes are set in the context of a definite model of high-energy physics. By default this is the Standard Model, dubbed `SM`.

Here is the SINDARIN code for defining a SM physics process, computing its cross section, and generating a simulated event sample:

```
process ee = e1, E1 -> e2, E2
compile

sqrts = 360 GeV
integrate (ee)

n_events = 10
?write_lhef = true
$file_lhef = "ee.lhef"
simulate (ee)
```

As before, you save this text in a file (named, e.g., `ee.sin`) which is run by

```
> whizard -r ee.sin
```

(We will come to the meaning of the `-r` option later.) This produces a lot of output. We break it down into pieces.

The startup is as before:

```
| Writing log to 'whizard.log'
|================================================================================|
|                               WHIZARD 2.0.0_rc1
|================================================================================|
| Initializing process library 'processes'
| Reading model file 'SM.mdl'
| Using model: SM
| Reading commands from file 'ee.sin'
| Added process to library 'processes':
|  [0] ee = e-, e+ -> mu-, mu+

| Generating code for process library 'processes'
| Calling O'Mega for process 'ee'
| command: /home/kilian/whizard/build/nagfor/src/omega/bin/omega_SM.opt -o ee.f90 -targe
[1/1] e- e+ -> mu- mu+ ... done. [time: 0.03 secs, total: 0.03 secs, remaining: 0.00 sec
all processes done. [total time: 0.03 secs]
SUMMARY: 6 fusions, 2 propagators, 2 diagrams
| Writing interface code for process library 'processes'
| Compiling process library 'processes'

| Loading process library 'processes'
| Process 'ee': updating previous configuration
sqrts =    3.6000000000000000E+02
| Integrating process 'ee'
| Generating phase space, writing file 'ee.phs' (this may take a while)
| Found 2 phase space channels.
Warning: No cuts have been defined.

| Using partonic energy as event scale.
| iterations = 3:1000, 3:10000
| Creating grids
| 1000 calls, 2 channels, 2 dimensions, 20 bins, stratified = T
|================================================================================|
| It      Calls  Integral[fb]  Error[fb]   Err[%]    Acc  Eff[%]   Chi2 N[It] |
|================================================================================|
   1       1000  8.3366006E+02  1.47E+00    0.18    0.06*  40.12
   2       1000  8.3357740E+02  8.16E-01    0.10    0.03*  40.11
   3       1000  8.3214263E+02  1.01E+00    0.12    0.04   57.40
```

```
|---------------------------------------------------------------------------|
   3       3000  8.3311382E+02  5.83E-01    0.07    0.04   57.40    0.69   3
|---------------------------------------------------------------------------|
   4      10000  8.3325834E+02  1.10E-01    0.01    0.01*  57.02
   5      10000  8.3333796E+02  1.11E-01    0.01    0.01   57.03
   6      10000  8.3323772E+02  1.11E-01    0.01    0.01   57.03
|===========================================================================|
   6      30000  8.3327798E+02  6.41E-02    0.01    0.01   57.03    0.23   3
|===========================================================================|


n_events =  10
?write_lhef -> true
$file_lhef -> "ee.lhef"
| No analysis setup has been provided.
| Writing events in LHEF format to file 'ee.lhef'
| Generating 10 events ...
| Writing events in internal format to file 'whizard.evx'
| ... done
| There were no errors and  1 warning(s).
| WHIZARD run finished.
|===========================================================================|
```

3.2 SINDARIN – The WHIZARD command language

3.2.1 Syntactic details of SINDARIN

In the SINDARIN language, there are certain pre-defined constructors or commands that cannot be used in different context by the user, which are – in alphabetical order – `$action`, `alias`, `all`, `$analysis_filename`, `and`, `as`, `any`, `beams`, `cmplx`, `combine`, `compile`, `cuts`, `$description`, `echo`, `else`, `exec`, `expect`, `false`, `?fatal_beam_decay`, `$file_debug`, `$file_default`, `file_hepmc`, `$file_lhef`, `if`, `include`, `int`, `integrate`, `iterations`, `$label`, `lhapdf`, `library`, `load`, `luminosity`, `model`, `n_events`, `no`, `observable`, `or`, `$physical_unit`, `plot`, `process`, `read_slha`, `real`, `?rebuild`, `?recompile`, `record`, `$restrictions`, `results`, `scan`, `seed`, `show`, `simulate`, `sqrts`, `then`, `$title`, `tolerance`, `true`, `unstable`, `write_analysis`, `?write_debug`, `?write_default`, `$write_hepmc`, `?write_lhef`, `write_slha`, `$xlabel`, and `$ylabel`. Also units are fixed, like `degree`, `eV`, `keV`, q `MeV`, `GeV`, and `TeV`. Again, these tags are locked and not user-redefinable. There functionality will be listed in detail below. Furthermore, a variable with a preceding question mark, ?, is a logical, while a preceding hash, #, denotes a character string variable. Also, a lot of unary and binary operators exist, `+ - \ , = : -> < > <= >= ^ () [] {} ~~~`, as well as quotation marks, ". Note that the different parentheses and brackets fulfill different purposes, which will be explained below. Comments in a line can be marked by a hash, #, or an exclamation mark, !.

- `$action`
  I have no real clue yet. In the example input file it is used to set an histogram manually by the individual record entries. But why is it a logical variable??? WK?

- `alias`
  This allows to define a collective expression for a class of particles, e.g. to define a generic expression for leptons, neutrinos or a jet as `alias lepton = e1:e2:e3:E1:E2:E3`, `alias neutrino = n1:n2:n3:N1:N2:N3`, and `alias jet = u:d:s:c:U:D:S:C:g`, respectively.

- `all`
  `all` is a function that works on a logical expression and a list, `all <log_expr> [<list>]`, and returns `true` if and only if `log_expr` is fulfilled for *all* entries in `list`, and `false` otherwise. Examples: `all Pt > 100 GeV [lepton]` checks whether all leptons are harder than 100 GeV, `all Dist > 2 [u:U, d:D]` checks whether all pairs of corresponding quarks are separated in R space by more than 2. Logical expressions with `all` can be logically combined with `and` and `or`. (cf. also `any`, `and`, `no`, and `or`)

- `$analysis_filename`
  This character variable allows to create a LaTeXfile for the user anaylsis, and to specify its name. If this variable is not set, the analysis will be directed to the screen output. (cf. also `write_analysis`)

- `and`
  This is the standard two-place logical connective that has the value true if both of its operands are true, otherwise a value of false. It is applied to logical values, e.g. cut expressions. (cf. also `or`).

- `as`
  cf. `compile`

- `any`
  `any` is a function that works on a logical expression and a list, `any <log_expr> [<list>]`, and returns `true` if `log_expr` is fulfilled for any entry in `list`, and `false` otherwise. Examples: `any PDG == 13 [lepton]` checks whether any lepton is a muon, `any E > 2 * mW [jet]` checks whether any jet has an energy of twice the W mass. Logical expressions with `any` can be logically combined with `and` and `or`. (cf. also `all`, `and`, `no`, and `or`)

- `beams`
  This specifies the contents and structure of the beams. If this command is absent in the input file, WHIZARD automatically takes the two incoming partons (or one for decays) of the corresponding process as beam particles and no structure functions are applied. Protons and antiprotons as beam particles are predefined as `p` and `pbar`, respectively. A structure function, like `lhapdf`, `ISR`, `EPA` and so on are switched on as e.g. `beams = p, p -> lhapdf`. (cf. also `circe`, `circe2`, `lhapdf`).

- `cmplx`
  Defines a complex variable. (to be finalized still

- `combine`
  The `combine [<list1>, <list2>]` operation makes a particle list whose entries are the result of adding (the momenta of) each pair of particles in the two input lists `list1`, list2. For example, `combine [incoming lepton, lepton]` constructs all mutual pairings of an incoming lepton with an outgoing lepton (an alias for the leptons has to defined, of course).

- `compile`
  The `compile` command is mandatory, it invokes the compilation of the process(es) (i.e. the matrix element file(s)) to be compiled as a shared library. This shared object file has the standard name `processes.so` and resides in the `.libs` subdirectory of the corresponding user workspace. If the user has defined a different library name `lib_name` with the `library` command, then WHIZARD compiles this as the shared object `.libs/lib_name.so`. (This allows to split process classes and to avoid too large libraries.) Another possibility is to use the command `compile as "static_name"`. This will compile and link the process library in a static way and create the static executable `static_name` in the user workspace. (cf. also `library`, `load`)

- `cuts`
  This command defines the cuts to be applied to certain processes. The syntax is: `cuts = <log_class> <log_expr> [<unary or binary particle (list) arg>]`, where the cut expression must be initialized with a logical classifier `log_class` like `all`, `any`, `no`. The logical expression `log_expr` contains the cut to be evaluated. Note that this need not only be a kinematical cut expression like `E > 10 GeV` or `5 degree < Theta < 175 degree`, but can also be some sort of trigger expression or event selection, e.g. PDG == 15 would select a tau lepton. Whether the expression is evaluated on particles or pairs of particles depends on whether the discriminating variable is unary or binary, `Dist` being obviously binary, `Pt` being unary. Note that some variables are both unary and binary, e.g. the invariant mass M. Cut expressions can be connected by the logical connectives `and` and `or`. The `cuts` statement acts on all subsequent process integrations and analyses until a new `cuts` statement appears. (cf. also `all`, `any`, `Dist`, `E`, `M`, `no`, `Pt`).

- `degree`
  Expression specifying the physical unit of degree for angular variables, e.g. the cut expression function `Theta`. (if no unit is specified for angular variables, radians are used).

- `$description`
  String variable that allows to specify a description text for the analysis, `$description = "analysis description text"`. This line appears below the title of a corresponding analysis, on top of the respective plot. (cf. `analysis`, `$title`)

- `echo`
  Allows to put verbose information on the screen during execution, e.g. `echo ("Hello, world!")`. (cf. also `show`)

- `else`
  cf. `if`

- `eV`
  Physical unit, stating that the corresponding number is in electron volt.

- `exec`
  Constructor `exec ("<cmd_name>")` that demands WHIZARD to execute/run the command `cmd_name`. For this to work that specific command must be present either in the path of the operating system or as a command in the user workspace.

- `expect`
  The binary function `expect` compares two numerical expressions whether they are fulfill a certain ordering condition or are equal up to a specific uncertainty or tolerance which can bet set by the specifier `tolerance`, i.e. in principle it checks whether a logical expression is true. The `expect` function does actually not just check a value for correctness, but also records its result. If failures are present when the program terminates, the exit code is nonzero. The syntax is `expect (<num1> <log_comp> <num2>)`, where `num1` and `num2` are two numerical values (or corresponding variables) and `log_comp` is one of the following logical comparators: $\langle$, $\rangle$, $\langle$`=`, $\rangle$`=`, `==`, `/=`, ~~, ~. (cf. also $\langle$, $\rangle$, $\langle$`=`, $\rangle$`=`, `==`, `/=`, ~~, ~, `tolerance`).

- `false`
  Constructor stating that a logical expression or variable is false, e.g. `?<log_var> = false`. (cf. also `true`).

- `?fatal_beam_decay`
  Logical variable that let the user decide whether the possibility of a beam decay is treated as a fatal error or only as a warning. An example is a process $bt \to X$, where the bottom quark as an inital state particle appears as a possible decay product of the second incoming particle, the top quark. This might trigger inconsistencies or instabilities in the phase space set-up.

- `$file_debug`
  String variable that allows via `$file_debug = "file_name"` to specify the name for the file `file_name` to which events in a a long verbose format with debugging information are written. If not set, the default file name is `whizard.debug`. (cf. also `?write_debug`)

- `$file_default`
  String variable that allows via `$file_default = "file_name"` to specify the name for the file `file_name` to which events in a human-readable format are written. If not set, the default file name is `whizard.default`. (cf. also `?write_default`)

- `$file_hepmc`
  String variable that allows via `$file_hepmc = "file_name"` to specify the name for the

file `file_name` to which events in the HepMC format are written. If not set, the default file name is `whizard.hepmc`. (cf. also `?write_hepmc`)

- `$file_lhef`
  String variable that allows via `$file_lhef = "file_name"` to specify the name for the file `file_name` to which events in the (new) Les Houches event (LHE) format (including XML headers) are written. If not set, the default file name is `whizard.lhef`. (cf. also `?write_lhef`)

- `GeV`
  Physical unit, energies in 10^9 electron volt. This is the default energy unit of WHIZARD.

- `if`
  Conditional clause with the construction `if <log_expr> then <expr> else <expr>`. Note that there is no specific `end if` statement. For more complicated expressions it is better to use expressions in parentheses: `if (<log_expr>) then {<expr>} else {<expr>}`. Examples are a selection of up quarks over down quarks depending on a logical variable: `if ?ok then u else d`, or the setting of an integer variable depending on the rapidity of some particle: `if (eta > 0) then { a = +1} else { a = -1}`. The `then` constructor is not mandatory and can be omitted.

- `include`
  The `include` statement, `include ("file.sin")` allows to include external SINDARIN files `file.sin` into the main WHIZARD input file. A standard example is the inclusion of the standard cut file `default_cuts.sin`.

- `int`
  This is a constructor to specify integer constants in the input file. Strictly speaking, it is a unary function setting the value `int_val` of the integer variable `int_var`: `int <int_var> = <int_val>`. (cf. also `real` and `cmplx`)

- `integrate`
  The `integrate (<proc_name>) { <integrate_options> }` command invokes the integration (phase-space grid generation and Monte-Carlo sampling of the process `proc_name` (which can also be a list of processes) with the integration options `<integrate_options>`. Right now the only option is to specify the number of iterations and calls per integration during the Monte-Carlo phase-space integration via `iterations = <n_iterations>:<n_calls>`. Note that this can be list, separated by colons, which breaks up the integration process into units of the specified number of integrations and calls each.

- `iterations`
  Option to set the number of iterations and calls per iteration during the Monte-Carlo phase-space integration process, cf. `integrate`.

- `keV`
  Physical unit, energies in 10^3 electron volt.

- `$label`
  This is a string variable, `$label = "label_name"` that allows to specify a label `label_name` for analysis plots on the x axis. It is only taken into account if the variable `$xlabel` has not been set, in which case it is overwritten by the string value of that variable. (cf. also `xlabel`, `ylabel`). WK: Do I see this correctly?

- `lhapdf`
  NOT YET PROPERLY WORKING!!!! (cf. `beams`)

- `library`
  The command `library = "<lib_name>"` allows to specify a separate shared object library archive `lib_name.so`, not using the standard library `processes.so`. Those libraries (when using shared libraries) are located in the `.libs` subdirectory of the user workspace. Specifying a separate library is useful for splitting up large lists of processes, or to restrict a larger number of different loaded model files to one specific process library. (cf. also `compile`, `load`)

- `load`
  The `load` command allows to load again a library if some details have been changed (processes added, redefined or maybe changed. Guess, here is some explanation missing (cf. also `compile`, `library`)

- `luminosity` This specifier `luminosity = <num>` sets the integrated luminosity for the event generation of the processes in the SINDARIN input files. Note that WHIZARD itself chooses the number from the `luminosity` or from the `n_events` specifier, whichever would give the larger number of events. As this depends on the cross section under consideration, it might be different for different processes in the process list. WK: Is this correct? Do we really want this? What about different units? Furthermore, the `luminosity` or `n_events` command has to be invoked *after* the corresponding logical variable which tells WHIZARD to write an event file in a specific format. (cf. `n_events`, `?write_debug`, `?write_default`, `$write_hepmc`, `?write_lhef`)

- `MeV`
  Physical unit, energies in 10^6 electron volt.

- `model`
  With this specifier, `model = <MODEL_NAME>`, one sets the hard interaction physics model for the processes defined after this model specification. The list of available models can be found in Table 5.1. Note that the model specification can appear arbitrarily often in a SINDARIN input file, e.g. for compiling and running processes defined in different physics models.

- `no`
  `no` is a function that works on a logical expression and a list, `no <log_expr> [<list>]`, and returns `true` if and only if `log_expr` is fulfilled for *none* of the entries in `list`, and

`false` otherwise. Examples: `no Pt < 100 GeV [lepton]` checks whether no lepton is softer than 100 GeV. It is the logical opposite of the function `all`. Logical expressions with `no` can be logically combined with `and` and `or`. (cf. also `all`, `any`, `and`, and `or`)

- `n_events`
  This specifier `n_events = <num>` sets the number of events for the event generation of the processes in the SINDARIN input files. Note that WHIZARD itself chooses the number from the `n_events` or from the `luminosity` specifier, whichever would give the larger number of events. As this depends on the cross section under consideration, it might be different for different processes in the process list. WK: Is this correct? Do we really want this? Furthermore, the `n_events` or `luminosity` command has to be invoked *after* the corresponding logical variable which tells WHIZARD to write an event file in a specific format. (cf. `luminosity`, `?write_debug`, `?write_default`, `$write_hepmc`, `?write_lhef`)

- `observable`
  With this, `observable = <obs_spec>`, the user is able to define a variable specifier `obs_spec` for observables. These can be reused in the analysis, e.g. as a `record`, as functions of the fundamental kinematical variables of the processes. (cf. `analysis`, `record`)

- `or`
  This is the standard two-place logical connective that has the value true if one of its operands is true, otherwise a value of false. It is applied to logical values, e.g. cut expressions. (cf. also `and`).

- `$physical_unit`
  This is a string variable, `$physical_unit = "<unit_name>''`, that allows to set a LaTeXname `unit_name` for the physical unit of a label of an analysis plot. This unit is then also used for calculations within the analysis set-up.

- `plot`

  (cf. `record`)

- `process`
  Allows to set a hard interaction process, either for a decay process `decay_proc` as `process <decay_proc> = <mother> -> <daughter1>, <daughter2>, ...`, or for a scattering process `scat_proc` as `<incoming1>, <incoming2> -> <outgoing1>, <outgoing2>, ...`. Note that there can be arbitrarily many processes to be defined in a SINDARIN input file. (cf. also `restrictions`)

- `read_slha`
  Tells WHIZARD to read in an input file in the SUSY Les Houches accord (SLHA), as `read_slha ("slha_file.slha")`. Note that the files for the use in WHIZARD should have the suffix `.slha`. (cf. also `write_slha`)

- `real`
  This is a constructor to specify real constants in the input file. Strictly speaking, it is a unary function setting the value `real_val` of the integer variable `real_var`: `real <real_var> = <real_val>`. (cf. also `int` and `cmplx`)

- `?rebuild`
  The logical variable `?rebuild = true/false` specifies whether the matrix element code for processes is re-generated by the matrix element generator O'Mega (e.g. if the process has been changed, but not its name). This can also be set as a command-line option `whizard --rebuild`. The default is `false`, i.e. code is never re-generated if it is present and the MD5 checksum is valid. (cf. also `recompile`).

- `?recompile`
  The logical variable `?recompile = true/false` specifies whether the matrix element code for processes is re-compiled (e.g. if the process code has been manually modified by the user). This can also be set as a command-line option `whizard --recompile`. The default is `false`, i.e. code is never re-compiled if its corresponding object file is present. (cf. also `rebuild`)

- `record`
  The `record` constructor provides an internal data structure in SINDARIN input files. Its syntax is in general `record <record_name> (<cmd_expr>)`. The `<cmd_expr>` could be the definition of a tuple of points for a histogram or an `eval` constructor that tells WHIZARD e.g. by which rule to calculate an observable to be stored in the record `record_name`. (cf. also `eval`)

- `$restrictions`
  This is an optional argument for process definitions. It defines a string variable, `process <process_name> = <particle1>, <particle2> -> <particle3>, <particle4>, ... { $restrictions = "<restriction_def>" }`. The string argument `restriction_def` is directly transferred during the code generation to the matrix element generator O'Mega. It has to be of the form `n1 + n2 + ... ~ <particle (list)>`, where `n1` and so on are the numbers of the particles above in the process definition. The tilde specifies a certain intermediate state to be equal to the particle(s) in `particle (list)`. An example is `process eemm_z = e1, E1 -> e2, E2 { $restrictions = "1+2 ~ Z" }` restricts the code to be generated for the process $e^-e^+ \to \mu^-\mu^+$ to the s-channel Z-boson exchange. (cf. also `process`)

- `results`
  Only used in the combination `show(results)`. Forces WHIZARD to print out a results summary for the integrated processes. (cf. also `show`)

- `scan`
  Constructor to perform loops over variables or scan over processes in the integration procedure. The syntax is `scan <var> <var_name> (<value list> or <value_init> ->`

<value_fin> /<incrementor> <increment>) { <scan_cmd> }. The variable var can be specified if it is not a real, e.g. an integer. var_name is the name of the variable which is also allowed to be a predefined one like seed. For the scan, one can either specify an explicit list of values value list, or use an initial and final value and a rule to increment. The scan_cmd can either be just a show to print out the scanned variable or the integration of a process. Examples are: scan seed (32 -> 1 / / 2) { show (seed_value) } , which runs the seed down in steps 32, 16, 8, 4, 2, 1 (division by two). scan mW (75 GeV, 80 GeV -> 82 GeV /+ 0.5 GeV, 83 GeV -> 90 GeV /* 1.2) { show (sw) } scans over the W mass for the values 75, 80, 80.5, 81, 81.5, 82, 83 GeV, namely one discrete value, steps by adding 0.5 GeV, and increase by 20 % (the latter having no effect as it already exceeds the final value). It prints out the corresponding value of the effective mixing angle which is defined as a dependent variable in the model input file(s). scan sqrts (500 GeV -> 600 GeV /+ 10 GeV) { integrate (proc) } . integrates the process proc in eleven increasing 10 GeV steps in center-of-mass energy from 500 to 600 GeV.

- seed
  Integer variable seed = <num> that allows to set a specific random seed num. If not set, WHIZARD takes the time from the system clock to determine the random seed.

- show
  This is a unary function that is operating on specific constructors in order to print them out in the WHIZARD screen output as well as the log file whizard.log. Examples are show(<parameter_name>) to issue a specific parameter from a model or a constant defined in a SINDARIN input file, show(integral(<proc_name>)), show(library), show(results), or show(¡var¿) for any arbitrary variable. (cf. also echo, library, results)

- simulate
  This command invokes the generation of events for the process proc by means of simulate (<proc>). (cf. also integrate, luminosity, n_events)

- sqrts
  Real variable in order to set the center-of-mass energy for the collisions (collider energy $\sqrt{s}$, not hard interaction energy $sqrt\hat{s}$): sqrts = <num> <phys_unit>. The physical unit can be one of the following eV, keV, MeV, GeV, and TeV. If absent, WHIZARD takes GeV as its standard unit.

- TeV
  Physical unit, for energies in 10^{12} electron volt.

- then
  Alternative option inside a conditional clause, not mandatory, hence maybe be omitted, cf. if.

```
process zee =   Z -> e1, E1
process zuu =   Z -> u, U
process zz = e1, E1 -> Z, Z
compile
integrate (zee) { iterations = 1:100 }
integrate (zuu) { iterations = 1:100 }
sqrts = 500 GeV
integrate (zz) { iterations = 3:5000, 2:5000 }
unstable Z (zee, zuu)
```

Figure 3.1: *SINDARIN input file for unstable particles and inclusive decays.*

- `$title`
  This string variable sets the title of a plot in a WHIZARD analysis setup, e.g. a histogram or an observable. The syntax is `$title = "<your title>"`. This title appears as a section header in the analysis file, but not in the screen output of the analysis. (cf. also `$description`, `$label`, `$xlabel`, `$ylabel`).

- `tolerance`
  Real variable that defines the tolerance with which the (logical) function `expect` accepts equality or inequality: `tolerance = <num>`. This can e.g. be used for cross-section tests and backwards compatibility checks. (cf. also `expect`)

- `true`
  Constructor stating that a logical expression or variable is true, e.g. `?<log_var> = true`. (cf. also `false`).

- `unstable`
  This constructor allows to let final state particles of the hard interaction undergo a subsequent (cascade) decay (in the on-shell approximation). For this the user has to define the list of desired Decay channels as `unstable <mother> (<decay1>, <decay2>, ....)`, where `mother` is the mother particle, and the argument is a list of decay channels. Note that these have to be provided by the user as in the example in Fig. 3.1. First, the Z decays to electrons and up quarks are generated, then ZZ production at a 500 GeV ILC is called, and then both Zs are decayed according to the probability distribution of the two generated decay matrix elements. This obviously allows also for inclusive decays.

- `write_analysis`
  The `write_analysis` statement tells WHIZARD to write the analysis setup by the user for the SINDARIN input file under consideration. If no `$analysis_filename` is provided, the analysis (including the histograms) are printed out on the screen, otherwise they are written to a file defined by that specific string variable. (cf. also `$analysis_filename`)

- `?write_debug`
  Logical variable that, if set true, demands WHIZARD to write out an event file in a long, verbose format for debugging purposes. Note that `simulate` has to be invoked for this in order to work as well as either a non-zero number of events, `n_events`, or a non-vanishing luminosity, `luminosity`. The `?write_debug` flag has to be set before the event or luminosity numbers! The standard name of the event file will be `whizard.debug`, but can be redefined by the `$file_debug` variable. (cf. `$file_debug`, `luminosity`, `n_events`, `simulate`)

- `?write_default`
  Logical variable that, if set true, demands WHIZARD to write out an event file in a standard ASCII format. Note that `simulate` has to be invoked for this in order to work as well as either a non-zero number of events, `n_events`, or a non-vanishing luminosity, `luminosity`. The `?write_default` flag has to be set before the event or luminosity numbers! The standard name of the event file will be `whizard.default`, but can be redefined by the `$file_default` variable. (cf. `$file_default`, `luminosity`, `n_events`, `simulate`)

- `$write_hepmc`
  Logical variable that, if set true, demands WHIZARD to write out an event file in the HepMC format (if externally linked). Note that `simulate` has to be invoked for this in order to work as well as either a non-zero number of events, `n_events`, or a non-vanishing luminosity, `luminosity`. The `?write_hepmc` flag has to be set before the event or luminosity numbers! The standard name of the event file will be `whizard.hepmc`, but can be redefined by the `$file_hepmc` variable. (cf. `$file_hepmc`, `luminosity`, `n_events`, `simulate`)

- `?write_lhef`
  Logical variable that, if set true, demands WHIZARD to write out an event file (new) Les Houches event format (LHEF, with XML headers). Note that `simulate` has to be invoked for this in order to work as well as either a non-zero number of events, `n_events`, or a non-vanishing luminosity, `luminosity`. The `?write_lhef` flag has to be set before the event or luminosity numbers! The standard name of the event file will be `whizard.lhef`, but can be redefined by the `$file_lhef` variable. (cf. `$file_lhef`, `luminosity`, `n_events`, `simulate`)

- `write_slha`
  Demands WHIZARD to write out a file in the SUSY Les Houches accord (SLHA). How this file gets its info is still completely unknown to me! Hard-coded right now? Docu is missing!!! (cf. also `read_slha`)

- `$xlabel`
  String variable, `$xlabel = "<LaTeX code>"`, that sets the x axis label in a plot or histogram in a WHIZARD analysis. (cf. also `label` and `$ylabel`)

- `$ylabel`
  String variable, `$ylabel = "<LaTeX code>"`, that sets the y axis label in a plot or histogram in a WHIZARD analysis. (cf. also `label` and `$xlabel`)

3.3 WHISH – The WHIZARD Shell/Interactive mode

WHIZARD can be also run in the interactive mode using its own shell environment. This is called the WHIZARD Shell (WHISH). For this purpose, one starts with the command

```
/home/user$ whizard --interactive
```

or

```
/home/user$ whizard -i
```

The WHISH can be closed by the `quit` command:

```
whish? quit
```

3.4 Using processes from several different models

When using two different models which need an SLHA input file, these *have* to be provided for both models. Otherwise WHIZARD will not be performing the phase-space setup for the second process.

Note that when using more than one models, the setting of parameters *after* the last model and process declarations only affects the active – i.e. the last – model. If one wants to set a parameter for all models in the input file, one has to repeat the model setting for every defined model. Athough this might seem cumbersome at first, it is nevertheless a sensible procedure since the parameters defined by the user might anyhow not be defined or available for all chosen models.

Chapter 4

Examples

Chapter 5

Implemented physics

5.1 The Monte-Carlo integration routine: VAMP

5.2 The Phase-Space Setup

5.3 The hard interaction models

5.3.1 The Standard Model and friends

5.3.2 Beyond the Standard Model

5.4 Technical details – Advanced Spells

5.4.1 Efficiency and tuning

Since massless fermions and vector bosons (or almost massless states in a certain approximation) lead to restrictive selection rules for allowed helicity combinations in the initial and final state. To make use of this fact for the efficiency of the WHIZARD program, we are applying some sort of heuristics: WHIZARD dices events into all combinatorially possible helicity configuration during a warm-up phase. The user can specify a helicity threshold which sets the number of zeros WHIZARD should have got back from a specific helicity combination in order to ignore that combination from now on. By that mechanism, typically half up to more than three quarters of all helicity combinations are discarded (and hence the corresponding number of matrix element calls). This reduces calculation time up to more than one order of magnitude. WHIZARD shows at the end of the integration those helicity combinations which finally contributed to the process matrix element.

Note that this list – due to the numerical heuristics – might very well depend on the number of calls for the matrix elements per iteration, and also on the corresponding random number seed.

MODEL TYPE	with CKM matrix	trivial CKM
Yukawa test model	---	`Test`
QED with e, μ, τ, γ	---	`QED`
QCD with d, u, s, c, b, t, g	---	`QCD`
Standard Model	`SM_CKM`	`SM`
SM with anomalous gauge couplings	`SM_ac_CKM`	`SM_ac`
SM with anomalous top couplings	---	`SM_top`
SM with K matrix	---	`SM_KM`
MSSM	`MSSM_CKM`	`MSSM`
MSSM with gravitinos	---	`MSSM_Grav`
NMSSM	`NMSSM_CKM`	`NMSSM`
extended SUSY models	---	`PSSSM`
Littlest Higgs	---	`Littlest`
Littlest Higgs with ungauged $U(1)$	---	`Littlest_Eta`
Littlest Higgs with T parity	---	`Littlest_Tpar`
Simplest Little Higgs (anomaly-free)	---	`Simplest`
Simplest Little Higgs (universal)	---	`Simplest_univ`
SM with graviton	---	`Xdim`
UED	---	`UED`
SM with Z'	---	`Zprime`
"SQED" with gravitino	---	`GravTest`
Augmentable SM template	---	`Template`

Table 5.1: *List of models available in WHIZARD. There are pure test models or models implemented for theoretical investigations, a long list of SM variants as well as a large number of BSM models.*

Acknowledgements

We would like to thank E. Boos, R. Chierici, K. Desch, M. Kobel, F. Krauss, P.M. Manakos, N. Meyer, K. Mönig, H. Reuter, T. Robens, S. Rosati, J. Schumacher, M. Schumacher, and C. Schwinn who contributed to `WHIZARD` by their suggestions, bits of codes and valuable remarks and/or used several versions of the program for real-life applications and thus helped a lot in debugging and improving the code. Special thanks go to A. Vaught and J. Weill for their continuos efforts on improving the g95 and gfortran compilers, respectively.

Bibliography

[1] T. Sjöstrand, Comput. Phys. Commun. **82** (1994) 74.

[2] A. Pukhov, *et al.*, Preprint INP MSU 98-41/542, `hep-ph/9908288`.

[3] T. Stelzer and W.F. Long, Comput. Phys. Commun. **81** (1994) 357.

[4] T. Ohl, *Proceedings of the Seventh International Workshop on Advanced Computing and Analysis Technics in Physics Research*, ACAT 2000, Fermilab, October 2000, IKDA-2000-30, `hep-ph/0011243`; M. Moretti, Th. Ohl, and J. Reuter, LC-TOOL-2001-040

[5] T. Ohl, Comput. Phys. Commun. **120** (1999) 13.

[6] T. Ohl, Comput. Phys. Commun. **101** (1997) 269.

[7] M. Skrzypek and S. Jadach, Z. Phys. **C49** (1991) 577.

[8] A. Djouadi, J. Kalinowski, M. Spira, Comput. Phys. Commun. **108** (1998) 56-74.

[9] E. Boos *et al.*, in: Proc. Les Houches 2001, `hep-ph/0109068`

[10] P. Skands *et al.*, arXiv:hep-ph/0311123.

[11] K. Hagiwara *et al.*, arXiv:hep-ph/0512260.

[12] B. C. Allanach *et al.*, in *Proc. of the APS/DPF/DPB Summer Study on the Future of Particle Physics (Snowmass 2001)* ed. N. Graf, Eur. Phys. J. C **25** (2002) 113 [eConf **C010630** (2001) P125] [arXiv:hep-ph/0202233].

[13] J. A. Aguilar-Saavedra *et al.*, arXiv:hep-ph/0511344.