The *SymbolicData* Benchmark Problems Collection of Polynomial Systems

`http://www.symbolicdata.org`

Hans-Gert Gräbe
Department of Computer Science
University of Leipzig, Germany
`graebe@informatik.uni-leipzig.de`

1 Introduction

1.1 Motivation

Authors of software or even packages or modules capable of special symbolic computations are soon or later faced with the problem to test and evaluate it. Whereas the 'copy and paste' method is usually sufficient for this aim during early development phases, reliable tests and screenings with large sets of data require batch processing and special test beds.

Such test bed environment should prepare test data for input to the CA software, start and monitor its run, and store and evaluate the output of the computation.

Although such test beds are mostly not related to the tested symbolic software (e.g., a Maple package) – and often use even different technologies (shell scripts and file redirection) – they are usually self-made (including special input formats for the examples) and, often enough, developed anew for each project.

To avoid this overhead it would be interesting to unify such efforts into a common project that collects experience with test beds and allows for easy reuse and modification of already existing code. The *SymbolicData* Project was set out to meet these demands.

Since easy reusability is best achieved by code under Free Software conditions and test beds are usually not written from scratch but reuse appropriate tools, we focussed on Free Software tools. At the moment our test bed

relies on Perl 5 and the GNU `time` function. A first release (mainly developed by O. Bachmann, Kaiserslautern, and the author) contains more than 40 Perl modules to handle, translate, sort, validate, and store different kinds of data. These packages (with more than 15 000 lines of code) are driven by a standard interface program `symbolicdata`. They are available from our Web site under the terms of the GNU Public License (GPL). Perl with its scripting and pattern matching facilities turned out to be best suited for the preparation of input data and starting and monitoring processes. The GNU time function provides an independent timing tool with the possibility to time out and interrupt processes.

A second (and historically even the first) motivation for the *SymbolicData* Project arose from questions related to comparison and benchmarking of symbolic software. Typical benchmark papers published so far report about own test computations that often enough could not be repeated by interested parties due to different reasons: the software is not available, huge examples are not supplied or supplied only in printed (and often misprinted) form, the authors refer to examples given in the literature in different (non equivalent) forms etc.

Such problems could be avoided if there was a central electronic repository storing benchmark examples in reliable formats that could easily be accessed by interested parties. Of course, this requires to give away own test material under Free Software conditions to the community. The *SymbolicData* Project started (with kindly acknowledged support by the French 'UMS Medicis' and the German 'Fachgruppe Computeralgebra') such data collections in the areas of polynomial system solving (to be reported below) and geometry theorem proving.

1.2 The *SymbolicData* Project – Aims and Current State

The main track followed so far with the *SymbolicData* project was to *develop a test bed* for symbolic software, to *systematically collect* existing special and general benchmark data and to make them *electronically available* in a more or less uniform way.

Note that symbolic computations often lead to voluminous data as input, output or intermediate results. Therefore, to collect benchmark data requires also to develop concepts and tools to generate, store, manipulate, present and maintain it.

Hence the *SymbolicData* project started with two main goals:

1. To unify efforts of several people to develop Perl tools for the management of digital symbolic data from different areas of Computer Algebra.

   These tools, although not yet perfect, are useful and can be adapted for special test and benchmark purposes at a local site. They are ready for download and improvement.

2. To provide a central repository of digital benchmark data from different areas of Computer Algebra.

   This repository at `http://www.symbolicdata.org` (it is sponsored by the German 'Fachgruppe Computeralgebra') contains the data collected so far and also provides access to the tools and documentation.

The project is organized as a free software project. The CVS repository is equally open to people joining the *SymbolicData* Project Group, and we enjoy your cooperation. Tools and data are freely available also as tar-files (via HTML download from our Web site) under the terms of the GNU Public License. The alpha release 0.4, available since March 2001, contains

- Tools to maintain digital symbolic data (below you will find a short overview),
- Digital data collections from the areas of polynomial system solving and geometry theorem proving,
- a well elaborated HTML documentation,
- and a small number of publications and presentations.

Due to the fact that the *SymbolicData* tools were used so far mainly for the management of symbolic non computational information, the release 0.5, available since January 2002, offers separately

- the *SymbolicData* tools with a minimal data collection (required to build the documentation)
- and the full *SymbolicData* data collection.

This new offer should be considered also by people who are interested to use our tools for local test or benchmark computations on their own data only.

1.3 The *SymbolicData* Tools

The *SymbolicData* tools developed so far are designed to meet three different goals:

1. To systematically collect and maintain digital benchmark data arising in various areas of Computer Algebra.

   The data is stored in a data base complying a XML-like syntax that easily may be extended and adapted. The *SymbolicData* Tools together with the flexibility of the Perl language allow to store, extract, combine, select, modify, present etc. data with various objectives in a unified way.

   The standard interface program `symbolicdata` can be used for the most common operations (insertion, validation, extension, update) without Perl knowledge. Due to the elaborated 'actions' concept it can be extended with little experience in Perl programming. Some examples are given below.

2. To facilitate test or even trusted benchmark computations on the collected data.

   The *SymbolicData* project provides concepts and tools to extract data from the data base in a form readable by different Computer Algebra Software, to set up, start, time, interrupt, and monitor computations on these input data, and to collect, analyze, and evaluate output data from these computations.

   This requires more flexibility and hence additional programming efforts by the user. We already designed several tools for a benchmark Compute environment, but this part of the project is yet under development.

3. To provide tools to access, select, translate and present data in different formats.

   This part of the project is rudimentary. *SymbolicData* temporary provides a small HTML interface for test purposes and an interface to SQL-compliant databases.

To contribute data to the repository or to join the *SymbolicData* group please consult our web site for more information.

2 The *SymbolicData* INTPS Collection

2.1 How Data are Organized

The *SymbolicData* data collection is designed using a relational data base model and stored in a XML-like ASCII format. This allows for easy manipulation and translation of this data in different formats.

Due to flexibility reasons we decided not to use (at least at the moment) one of the various data base programs as main engine but implemented a Perl interface to access and manipulate data. Data records are stored as files (**sd-files**) and attached to the Perl interface in a transparent way as records of tag/value pairs (**sd-records**) using Perl 5 modular technology.

Similar records share a common structure and are grouped into **tables**. Tables correspond to subdirectories of the Data directory tree. The main information about benchmark collections of polynomial systems is contained in the `INTPS` table. According to the relational data base model secondary information about these records is scattered over several other tables (`BIB` for bibliographical references, `PROBLEMS` for problem or problem class descriptions, `GEO` for geometry theorem proving background of relevant `INTPS` records etc.) and linked with the main record through a `CRef` attribute.

A typical `INTPS` sd-file, Trinks' example, see [3], is reproduced on the next page. For a description of the different attributes see below.

2.2 Cross References

You may ask for more information about this example, e.g., bibliographical references. Such relational information combines two records and, in a relational data base model, it is usually stored in special relation tables that can easily be searched for different keys. We decided to put this cross reference information into one of the main (primary) records and to provide tools to extract it as secondary data in SQL compliant form for import into relation tables of a classical data base engine with search and select facilities. This avoids to develop anew elaborated search and select facilities for the primary (XML based) data.

For Trinks' example, relational bibliographical information is stored in `BIB` table records and uses the Trinks' example's `Id` as foreign key. Below the `BIB` record of [3] is reproduced.

The main reason for the decision to declare the `INTPS` table as foreign is persistence in the sense that we do not need to change an `INTPS` record each time a new publication refers to it. For analogous reasons the `BIB` table is declared as foreign in a `CRef` entry in some `INTPS` records that point to

```
######################################################################
# Record 'INTPS/Trinks'

<Id>        INTPS/Trinks                    </Id>
<Type>      INTPS                           </Type>
<Key>       Trinks                          </Key>
<basis>
[
  35*p+40*z+25*t-27*s,
  45*p+35*s-165*b-36,
  -11*s*b+3*b^2+99*w,
  25*p*s-165*b^2+15*w+30*z-18*t,
  15*p*t+20*z*s-9*w,
  -11*b^3+w*p+2*z*t
]
</basis>
<vars>      [w, p, z, t, s, b]              </vars>
<dlist>     [1, 1, 2, 2, 2, 3]              </dlist>
<isHomog>   0                               </isHomog>
<llist>     [4, 4, 3, 5, 3, 3]              </llist>
<degree>    10                              </degree>
<Comment>   diff = easy                     </Comment>
<Version>   ...                             </Version>
<PERSON>    graebe                          </PERSON>
<Date>      Mar 26 1999                     </Date>

# End of record 'INTPS/Trinks'
######################################################################
```

the primary source where the polynomial system was mentioned first time. Note that it is not always as easy as here to make such a judicious decision. Secondary data may be searched with an SQL compliant data base engine for both the primary and the foreign keys.

2.3 The Structure of `INTPS` Records

Following the XML philosophy the attributes of records (i.e., the XML tag names) and their descriptions are not fixed within the *SymbolicData* tools but are part of the data. Due to lacking experience we did not use DTD and XSL style sheets at the moment to describe tag syntax and semantics but collected this information in special META records and developed Perl tools to extract the descriptions from these META tables.

```
######################################################################
# Record 'BIB/Boege_86a'

<Id>        BIB/Boege_86a                  </Id>
...
<bibentry>
@Article{Boege_86a,
  author =       {Boege, W. and Gebauer, R. and Kredel, H.},
  title =        {Some examples for solving systems of algebraic
                  equations by calculating {Gr\"obner} bases},
  journal =      {J. Symb. Comp.},
  volume =       {2},
  year =         {1986},
  pages =        {83 - 98},
}
</bibentry>
...
<CRef>
[
  INTPS/Hairer_1 => Hairer 1,
  INTPS/Hairer_2 => Hairer 2,
  ...
  INTPS/Rose => Rose,
  INTPS/Trinks => Trinks,
  INTPS/Trinks_1 => Small Trinks
]
</CRef>
...
# End of record 'BIB/Boege_86a'
######################################################################
```

This allows for great flexibility and careful design of data tables by users. Templates are easily created, extended or changed varying the corresponding META tables with your favorite text editor.

Designing the structure of `INTPS` records we tried to specify a framework that unifies the different benchmark collections of systems of polynomials as, e.g., [2, 3, 4, 7, 8, 9]. Each such system of polynomials is defined through a finite basis in a certain polynomial ring $R[\mathbf{x}]$ in a list of variables $\mathbf{x}$ over a base domain R. It occurs that most examples may be reduced to systems of polynomials with integer coefficients or with coefficients in $R = \mathbf{Z}[\mathbf{p}]$ where $\mathbf{p}$ is a list of parameters. We decided to focus on such systems.

For uniformity reasons and to ease comparison, we require of a valid

`INTPS` record, that its basis polynomials are stored in expanded standard form using the `+`, `*`, and `^` operators, and that the monomials of a polynomial and the polynomials of the basis are ordered w.r.t. the degree reverse lexicographical ordering. The *SymbolicData* `Validate` action can fix these properties of an `INTPS` record if you have SINGULAR [6] installed on your computer[1].

Further tags are defined to collect background information about the different polynomial systems. Background information may be of structural or relational type. Structural information about a polynomial system concerns invariant properties of the basis and the ideal generated by it, e.g., lists of the lengths and degrees of the basis polynomials, the dimension or degree of the ideal, a prime or primary decomposition of the ideal, or certain parameters of such a description. Several optional tags, like `llist`, `dlist`, `dim`, `degree`, `isoPrimes`, `isoPrimeDims`, etc., and Perl routines are defined to collect or even generate such information.

The mandatory and optional attributes of `INTPS` records are listed in the table on the next page. Their structure and semantics is stored in a special `META` table `META/INTPS` in the same XML-format as the records themselves and thus may easily be extended or modified if necessary. The `META` tables are part of the Data directory tree and read in by the Perl tools during initialization.

3 The *SymbolicData* Perl Tools

SymbolicData provides a great variety of tools to perform operations on the collected data. These tools are of very different nature and requirements: they range from the insertion and validation of single records, over the initiation, control and evaluation of test or benchmark computations on selected lists of records, up to the transformation of parts or the entire data base into other representations like HTML or SQL.

The operations are implemented in a hierarchy of **Perl modules** and can be accessed in a unique way as **actions** invoked through the `symbolicdata` program that provides a standard interface and realizes command-line parsing, initialization of global variables and required modules, and execution of the actions inherited from the command line.

It is easy to add new functionality to the program since actions are stored as a global hash `$ACTIONS` that may be extended by new entries.

[1] There is a stub to use also other CAS for this purpose, but no implementation yet for other systems.

`Id`, `Type` and `Key` (m)
: Strings that identify the record within the data base (`Id` = `Type` + `Key` is generated automatically).

`basis` (m)
: A list of polynomials in expanded standard form with integer coefficients, defining an ideal I.

`vars` (m) and `parameters` (o)
: Lists of variables. I is considered as an ideal in the polynomial ring $R = k(\texttt{param})[\texttt{vars}]$ where k denotes the basic coefficient field.

`basedomain` (o)
: The basic coefficient field (default: $\mathbf{Q}$).

`dlist`, `llist` (o)
: Lists of total degrees and lengths of the basis elements. This gives rough invariants to identify records containing the same basis in different variable notations.

`attributes`, `dim`, `isHomog`, `IsoPrimes`, ...
: More information about I if available.

`ChangeLog`, `Version`, `PERSON`, `Date`
: Information about the history of the record and the person who supplied the information. There is a special table `PERSON` that collects more information about the people involved with *SymbolicData* and keeps historical track of their activities.

`CRef`
: A list of cross references to related records in other tables.

Mandatory (m) and optional (o) attributes of `INTPS` records

Below we give some examples of user defined actions. Consult the *SymbolicData* documentation and the source of the module `ActionsSpec.pm` for more details.

The overall syntax of a `symbolicdata` call is

```
symbolicdata [-r file] actions [options] [args]
```

On start-up `symbolicdata` loads all basic modules, parses the command-line arguments up to the mandatory action argument(s), and loads the global action hash which specifies all known (or, "registered") actions and their properties, e.g., the Perl modules required for the action, a description of the action etc. The action hash can be extended using the first (optional) `-r`

`file` argument, where `file` is the name of a Perl module which is loaded *before* the actions are parsed.

Note that some parts even of the basic features are yet under development, e.g., the search and find facilities of *SymbolicData*.

Different operations on the data require different degrees of flexibility. For example, starting test or benchmark computations on a special CAS requires translation of the data into the special input format of the tested system and hence some Perl programming. The *SymbolicData* actions concept is best suited to write such extensions almost from scratch into a file and get them running with

```
symbolicdata -r file ...
```

See the directory `bin/scripts` for sample extensions. We come back to that question in the next section.

A number of "standard" actions, mainly for insertion and validation of new records and extraction of SQL-compliant cross reference information are directly available through the `symbolicdata` interface. Here is a (not complete) list of such actions:

Manipulation of data base entries:

Insert	Insert sd-record into DataBase
Validate	Validate sd-record(s)
Update	Update sd-record in DataBase from foreign source
Unique	Test for uniqueness of sd-record(s) w.r.t. DataBase
Out	Print records to STDOUT
Print	Print fields of sd-record(s) to STDOUT

Creation of new INTPS records:

CreateINTPS	Create a new INTPS record from a GEO record of equational type
Flat	Generate a new INTPS record with flat basis from an INTPS record with parameters
Homog	Generate a new INTPS record with homogenized basis from an inhomogeneous INTPS record

Evaluation of information in BIB records:

GetAllBibs	extract BiBTeX entries from all BIB records to STDOUT
MakeBib	Create a file A`.bib` from A`.aux` and relevant BIB records

Extraction of CRef information:

CreateSQL	extract SQL table definitions to STDOUT
UpdateSQL	Print update information for SQL tables to STDOUT

Actions can be driven by various options. Please consult the documentation for more details. E.g., to generate/update the SQL cross reference information table for some of the INTPS records (with Key matching `Sym*`) issue the command (in the *SymbolicData* home directory)

```
symbolicdata UpdateSQL -Table CRefTable Data/INTPS/Sym*.sd
```

`CRefTable` is a predefined SQL table (also stored as sd-file in the directory `Data/SQL`) to catch cross reference information. You get a listing like

```
delete from CRefTable where Id='INTPS/Sym1_211';
insert into CRefTable values('INTPS/Sym1_211','PROBLEMS/Sym1','');
delete from CRefTable where Id='INTPS/Sym1_311';
insert into CRefTable values('INTPS/Sym1_311','PROBLEMS/Sym1','');
...
delete from CRefTable where Id='INTPS/Sym3_5';
insert into CRefTable values('INTPS/Sym3_5','PROBLEMS/Sym3','');
```

The result may be piped to a database program (we used Postgres95) to update the CRefTable created earlier with the command

```
symbolicdata CreateSQL -Table CRefTable
```

that yields output

```
create table CRefTable (
Id           varchar(80) not null,
Foreign_Id   varchar(80) not null,
Comment      varchar(100));
```

4 How to Run Local Benchmark Computations

A first series of benchmark computations on `INTPS` records was designed and executed by Olaf Bachmann in the year 2000. He developed the `Compute` Perl module that realizes computations as an elaborated interplay between configurations of Computer Algebra Software (table `CASCONFIG`), machines (table `MACHINE`) and examples (table `INTPS`). See the paper [1] for more details about this concept.

There was no continuation of these efforts when Olaf left the project team and this part remains experimental still now.

But it is easy to set up local benchmark computations also without an elaborated environment if you have data available in electronic form and the Perl scripting facilities at hand to create CAS input files and analyze output files.

As an example we consider benchmark computations to test the `solve` facility of MuPAD on zero dimensional ideals as described in [5]. To set up such computations we create a file `scripts/Compute.pl` that defines a new action `SolveTest`. This action is called via the `symbolicdata` interface program as

`symbolicdata -r "scripts/Compute.pl" SolveTest [`*sd-files*`]`

`symbolicdata` parses the input line, expands the sd-file names and calls the action on each of the sd-records. Hence the most difficult part of an action definition is the `call` slot that contains a Perl function to be executed on the corresponding sd-record.

In our example this Perl function creates an input file `/tmp/`*Key*`.in` that contains the MuPAD code of the example and starts a system call

```
mupad <$infile >$outfile 2>&1
```

via `TimedSystem`. `TimedSystem` is a special *SymbolicData* Perl function defined in the module `TimedSystem.pm` that allows to time and trap a computation. It is based on the GNU time function. We refer to the *SymbolicData* online documentation for more details.

Such an approach possibly does not meet your needs since it includes for each example the come up time of the CA software. An alternative solution uses the (system dependent) inner time function (e.g., MuPAD's `traperror` function) to time computations and is described below.

On the next page you find a listing of `Compute.pl` for the solution with GNU time. Note that different keys of an actions hash entry may carry also verbose (key 'verbose') and usage information (key 'example') and even a detailed HTML description (key 'description') about the action.

The function `thecomputation` extracts the relevant values from the sd-record and arranges them as MuPAD input lines (the code between the EOT's). During execution of the action on that record this code is written to a file `/tmp/`*Key*`.in`. Then MuPAD is started with a time bound of 100 s. to solve the problem. Upon success the output of the computation is written to a file `/tmp/`*Key*`.out` that can be analyzed either by hand or with additional Perl functions.

```
$ACTIONS -> {SolveTest} =
{
 verbose => "Benchmark computations with MuPAD and TimedSystem",
 req => ['TimedSystem.pm'],
 call => sub
 {
   my $r=shift;
   # create the infile
   my $infile="/tmp/$r->{Key}.in";
   open(FH,">$infile") or
        die "Can't open $infile for writing: $!\n";
   print FH thecomputation($r);
   close(FH);
   # set up the computation
   my $outfile="/tmp/$r->{Key}.out";
   my $maxtime=100;
   my $syscall="mupad <$infile >$outfile 2>&1";
   # start the computation
   my @l=TimedSystem($syscall,$maxtime,0,0);
   # evaluate the computation
   if ($l[0]<0)
   { print("$r->{Key} not finished within $maxtime sec.\n"); }
   else
   { printf("$r->{Key}: user time %1.2f, system time %1.2f.\n",
            $l[1], $l[2]); }
   return $r;
 },
 example  => 'symbolicdata -r "$SD_HOME/bin/scripts/Compute.pl" '
             .' SolveTest $SD_HOME/Data/INTPS/Sym1_211.sd',
};

sub thecomputation
{
  my $r=shift;
  my $s=<<EOT;
PRETTYPRINT:=FALSE;
vars:=$r->{vars};
polys:=$r->{basis};
tt:=time((sol:=solve(polys,vars))); sol; nops(sol);
tt:=time((sol1:=numeric::solve(polys,vars))); sol1; nops(sol1);
tt:=time((sol2:=map(sol,op\@allvalues))); sol2; nops(sol2);
quit;
EOT
  return $s;
}
```

The code itself is straightforward for slightly experienced Perl programmers and will not be discussed here.

For a solution using MuPAD's `traperror` function instead of GNU time use the *SymbolicData* tools to generate an appropriate input file, run it separately with MuPAD and inspect the results. Here is the relevant Perl code for a new action `TrapTest`:

```
$ACTIONS -> {TrapTest} =
{
 verbose => "Benchmark computations with MuPAD and traperror",
 argvcall => sub
 {
   shift; my $arg=ExpandArgv(shift);
   my $l;
   map push(@$l, Record->new($_)), (@$arg);
   # create the infile
   my $infile="/tmp/mupad.in";
   open(FH,">$infile") or
        die "Can't open $infile for writing: $!\n";
   print FH inittext();
   map { print FH trapcomputation($_); } (@$l);
   print FH exittext();
   close(FH);
   print "Input file written to $infile\n";
 },
 example  => 'symbolicdata -r "$SD_HOME/bin/scripts/Compute.pl" '
        . ' TrapTest $SD_HOME/Data/INTPS/S*.sd',
};
```

The first lines collect the sd-records to be tested from their (expanded) file names. Then we create the (single) input file `/tmp/mupad.in` containing the different examples. This requires some additional code, mainly for the function `trapcomputation`, reproduced on the next page.

Now start the test computation as

```
mupad </tmp/mupad.in >/tmp/mupad.out 2>&1
```

5 Extending the Data Base

In a similar fashion the data base may be extended to incorporate new examples even from new application areas. We document a first scratch extension to examples from Integer Programming that arose from a conversation with Raymond Hemmecke, who runs the Web site `http://www.testsets.de`.

```
sub inittext { return "PRETTYPRINT:=FALSE;\n"; }
sub exittext { return "quit;\n"; }

my $time=10;
sub trapcomputation
{
  my $r=shift;
  my $s=<<EOT;
// Example $r->{Key}
vars:=$r->{vars};
polys:=$r->{basis};
delete sol, sol1, sol2;
traperror((sol:=solve(polys,vars)),$time); sol;
traperror((sol1:=numeric::solve(polys,vars)),$time); sol1;
traperror((sol2:=map(sol,op\@allvalues)),$time); sol2;

EOT
  return $s;
}
```

Auxiliary Perl code for the TrapTest action

Given an integer-valued matrix A with n columns one may ask the challenging questions to compute the Hilbert basis or the extremal rays of the cone $\{x \in \mathbf{Z}^n : \ A\,x = 0,\ x \geq 0\}$.

Hemmecke's data collection contains files A`.mat`, A`.hil`, A`.ray` for each example A with lists of integer-valued vectors, one per line. The integer values are separated by white spaces. A first line gives the dimensions of the matrix.

To insert records of the new application into the *SymbolicData* data base we define a new table `TESTSETS`, i.e., create such subdirectories of `Data` and `Data/META`. Beside standard attributes (`Key, Type, PERSON, Date`, ...) already defined in the `Data/META` root directory each new record should have a mandatory attribute `mat` for the matrix A and optional attributes `hil` and `ray` for the Hilbert basis and the list of extremal rays. Values for the latter attributes are optional since either their computation may be to challenging or the output to heavy. For the latter case we create an (optional) attribute `file` to store the location of the corresponding file at `www.testsets.de`. For the moment matrices will be stored as lists of vectors in Hemmecke's format skipping the (redundant) dimension information. Instead we define another mandatory attribute `dim` with the ambient space dimension as value.

To generate a new table with these attributes one has to create META sd-records `dim`, `mat`, `hil`, `ray`, and `file`, i.e., files `dim.sd`, `mat.sd`, `hil.sd`, `ray.sd`, and `file.sd` in the `Data/META/TESTSETS` directory. A typical such META sd-file is reproduced below.

```
#################################################################
# Record 'META/TESTSETS/mat'

<Id>          META/TESTSETS/mat              </Id>
<Type>        META                           </Type>
<Key>         TESTSETS/mat                   </Key>
<Syntax>      (-|\d|\s)*                     </Syntax>
<description> Generating set of vectors      </description>
<level>       1                              </level>
<Version>     ...                            </Version>
<PERSON>      graebe                         </PERSON>
<Date>        Jan 18 2002                    </Date>

# End of record 'META/TESTSETS/mat'
#################################################################
```

META sd-files can be created with your favorite text editor starting with a copy of a META sd-file from another directory as template. Level one indicates mandatory tags, level greater one optional tags (default is 3). The `Syntax` Meta attribute that defines a valid syntax of `mat` values is given in Perl regexp notation. Take `.*` to pose no restrictions.

Now you can add new records to the `TESTSETS` table. This can be realized by another action written from scratch:

```
$ACTIONS -> {Create} =
{
 verbose => "Create TESTSETS from *.mat files",
 argvcall => sub
 {
   shift; my $arg=shift;   # get remaining args
   map createNewRecord($_), grep(/\.mat$/, @{$arg});
 },
 example  => 'symbolicdata -r "$SD_HOME/bin/scripts/testsets.pl" '
          . ' Create <files> ',
};
```

It is part of a file `testsets.pl` and creates new records from files `*.mat` and stores them in sd-files in the temporary directory `/tmp`. Values for the other attributes can be added in later steps (actually, a slight extension of

`testsets.pl` grasps also these values). `createNewRecord` is a user defined Perl function that creates sd-files with the desired content in a temporary directory.

In a second step these files are inserted into the *SymbolicData* data base with the command

```
symbolicdata Insert -fix /tmp/*.sd
```

This will validate the new records, generate (as far as possible) and insert missing tag values, format the output nicely and store it in the data base according to the `Id` tag value. Hence the actual file name of the temporary file does not matter. We use increasing numbers as file names:

```
my $i=0;

sub createNewRecord
{
  local $/;
  my $fn=shift;
  my ($r,$a);

  # set Key and Type
  ($r->{Key}=$fn)=~s/\.mat$//;
  $r->{Type}="TESTSETS";

  # evaluate *.mat
  open(FH,$fn) or die;
  $_=<FH>;
  my @l=split /\s*\n/ ; my $u=shift @l;
  my @l1=split(/\s+/,$u);
  $r->{dim}=shift @l1;
  $r->{mat}=join("\n", @l);
  close FH;

  # output the result
  $r=Record->new($r);
  $r->Out("/tmp/".$i++.".sd");
}
```

`Record->new` blesses `$r` to a sd-record and `$r->Out` writes it to the desired location. Missing attribute values (of `Id`, `Date` and `PERSON`) are generated during insertion.

6 How to Locally Install the Tools and Data

You may download the tools, data and documentation of *SymbolicData* as zipped Tar-files `SD-tools.tgz` and `SD-data.tgz` from our central repository at `http://www.symbolicdata.org`.

To work with the tools of *SymbolicData* you must have Perl version 5 (or higher) installed on your system.

To install the *SymbolicData* software and data, run GNU tar

```
tar -xzf SD-tools.tgz
tar -xzf SD-data.tgz
```

This will create a directory `SymbolicData` with several subdirectories containing the Perl tools, data, and documentation sources of the *SymbolicData* project.

Set the environment variable `SD_HOME` to that directory, change to it and run GNU make

```
make all
```

to create a new directory `SD_HTML` and generate the HTML documentation from their sources at this location. This is also a first test for the *SymbolicData* tools to be properly installed.

We refer to the `SymbolicData/README` file and the *SymbolicData* HTML documentation for further details.

References

[1] O. Bachmann and H.-G. Gräbe. The *SymbolicData* Project: Towards an electronic repository of tools and data for benchmarks of computer algebra software. Reports on Computer Algebra 27, Jan 2000. Centre for Computer Algebra, University of Kaiserslautern.
See `http://www.mathematik.uni-kl.de/~zca`.

[2] D. Bini and B. Mourrain. Polynomial test suite, 1996.
See `http://www-sop.inria.fr/saga/POL`.

[3] W. Boege, R. Gebauer, and H. Kredel. Some examples for solving systems of algebraic equations by calculating Gröbner bases. *J. Symb. Comp.*, 2:83 – 98, 1986.

[4] S.R. Czapor and K.O. Geddes. On implementing Buchberger's algorithm for Gröbner bases. In *Proc. SYMSAC'86*, pages 233 – 238. Waterloo, Canada, 1986.

[5] H.-G. Gräbe. About the polynomial system solve facility of Axiom, Macsyma, Maple, Mathematica, MuPAD, and Reduce. In M. Wester, editor, *Computer Algebra Systems: A Practical Guide*, chapter 8, pages 121 – 151. Wiley, Chichester, 1999.

[6] G.-M. Greuel, G. Pfister, and H. Schönemann. SINGULAR 2.0. A Computer Algebra System for Polynomial Computations, Centre for Computer Algebra, University of Kaiserslautern, 2001.
`http://www.singular.uni-kl.de`.

[7] PoSSo: Polynomial System Solving, 1993 – 1995.
See `http://posso.dm.unipi.it`.

[8] D. Wang. Irreducible decomposition of algebraic varieties via characteristic sets and Gröbner bases. *Computer Aided Geometric Design*, 9:471 – 484, 1992.

[9] D. Wang. Solving polynomial equations: characteristic sets and triangular systems. *Math. and Comp. in Simulation*, 42:339 – 351, 1996.