The SymbolicData Project

Towards an Electronic Repository of Tools and Data for Benchmarks of Computer Algebra Software

Olaf Bachmann
Department of Mathematics
University of Kaiserslautern, Germany
obachman@mathematik.uni-kl.de

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

http://www.SymbolicData.org[*]

Abstract

The SymbolicData project has the following three main goals: 1. to systematically collect existing symbolic computation benchmark data and to produce tools to extend and maintain this collection; 2. to design and implement concepts for trusted benchmarks computations on the collected data; and 3. to provide tools for data access/selection/transformation using different technologies.

SymbolicData has developed from a "grass root initiative" of a small number of people to a stage where it should be presented to, and evaluated and used by a wider community.

In this paper we report about the current state of the project, i.e., we describe the main design principles and tools which were developed to realize our goals.

1 Introduction

For different purposes, computer hardware and software is often tested on certain benchmarks. Although being sometimes controversially discussed, such benchmarks set (at least) well defined environments to compare otherwise incomparable technologies, algorithms, and implementations.

Benchmark suites for symbolic computations are not as well established as for other areas of computer science. This is probably due to the fact that there are not yet well agreed upon aims and technologies of such a benchmarking. However, during the last years efforts towards systematic benchmark collections for symbolic computations were intensified.

Following the trend of the development of Computer Algebra software, we can classify these efforts roughly into two categories:

1. *General* benchmarks which cover almost all areas of symbolic computation and whose main intend is to compare general-purpose Computer Algebra systems (CAS). The famous Wester suite [13, ch.3], is a typical example of such an effort.

2. *Special* benchmarks which concentrate only on a particular problem and whose main intend is to compare algorithms and implementations solving this problem. There are numerous special benchmarks for many particular problems scattered through the literature. See, e.g., [1, 2, 4, 8, 11, 12] for benchmarks of polynomial systems solving or [10, 14] for the polynomial factorization challenge.

For further qualification of these efforts it would be of great benefit to *unify* the different benchmark approaches and to *systematically collect* the existing special and general benchmark data such that they are *electronically available* in a more or less uniform way. This would provide the community with an electronic repository of certified inputs and results that could be addressed and extended during further development. The SymbolicData project is set out to realize this.

However, the aims mentioned above do not reach far enough: symbolic computations often lead to voluminous data as input, output or intermediate results. Therefore, such a project has not only to collect benchmark data but also to develop tools to generate, store, manipulate, present and maintain it.

Consequently, the SymbolicData project has the following three goals:

1. To systematically collect existing symbolic computation benchmark data and to produce tools with which this data collection can conveniently be extended and maintained.

2. To design and implement concepts which facilitate trusted benchmarks computations on the collected data.

3. To provide tools that allow data access/selection using different technologies (ASCII parser, SQL, WWW, etc) and data conversions into commonly used formats, e.g., HTML, SQL data bases, ASCII, LaTeX, etc.

In the first development stage of the project we concentrated on the general design principles of the tools and the data collection, thereby trying to achieve a balance between the necessary flexibility/extensibility on the one hand, and simplicity/practicability on the other.

A first application of our tools and concepts was realized on collections of data from two areas of Computer Algebra: Polynomial System Solving and Geometry Theorem Proving.

[*] At the time of the submission of this paper, the registration of this domain was not yet completed. In the mean-time, a mirror of what is to appear at this domain can be reached at http://www.informatik.uni-leipzig.de/~graebe/SymbolicData

Further applications of our tools and concepts to collect data from other areas of symbolic computation are intended. For this, we seek the cooperation of persons and groups that have related data collections at their disposal and are willing to spend some effort to enter these data into the SymbolicData data base and provide the respective add-ons to already existing tools.

The SymbolicData project grew out of the special session on benchmarking at the 1998 ISSAC conference in Rostock which was organized by H. Kredel. Since then, the project has steadily developed from ideas to implementations and data collections and back. At the beginning of 1999, the authors joint forces with the symbolic computation groups of the University of Paris VI (J. C. Faugere, D. Lazard), of Ecole Polytechnique (J. Marchand, M. Giusti), and of the University of Saarbrücken (M. Dengel, W. Decker). Furthermore, the project was incooperated into the benchmarking activities of the Fachgruppe Computeralgebra of the Deutsche Mathematiker Vereinigung.

In this paper we report about the current state of the SymbolicData project. Based on the general design of SymbolicData which is outlined in section 2, we describe in section 3 how the above mentioned goals were realized. These concepts are illustrated in section 4 by two examples of data collections from different areas of Computer Algebra. Section 5 gives an overview of what deliverables the SymbolicData project has produced so far which is finally followed by some concluding remarks in section 6.

2 The Design of SymbolicData

Based on the goals mentioned above and on the observation that the data to be collected enjoys a lot of structure, we choose an object-relational data base approach for the realization of SymbolicData. This approach does not only allow to systematically collect and store data, but also offers concepts to interrelate different data, e.g., problem descriptions, computational results, background information, citations, and to design modular, object-oriented tools for data access and manipulations.

For flexibility reasons, we do not use (at least at the moment) one of the various data base programs as main engine but keep the primary sources in an XML-like ASCII format. A file stored in a flat, XML-like syntax is well suited for direct editing and viewing, and for retrieving its information as a record of tag/value pairs combined from the tag name and the string enclosed between the (consecutive top level) start/end tags as value. We call such files **sd-files** and their associated records **sd-records** and use them as the basic units to store all information.

Furthermore, we use Perl as the programming language in which almost all of the tools for accessing and manipulating sd-records are written. Perl with its powerful scripting and string manipulation facilities, and its capability to design and implement modular and object-oriented tools turned out to be very adequate for this task.

2.1 The structure of the data base

As mentioned above, sd-records (or, records, for short) form the informational units of the data base and contain, e.g., problem descriptions, examples, references to the literature etc. Similar records share a common structure and are grouped into **tables**. Each sd-record must have a `Type` tag whose value specifies the table the record belongs to and a `Key` tag which uniquely identifies the record within its table.

There are two basic kinds of tables: data tables and meta tables. Data tables are used to actually store the collected data whereas meta tables are used to specify and define the syntax and semantics of the tags of data tables. More precisely, for each known tag of a particular data table, there is a sd-record in the corresponding meta table which specifies a set of **attributes** of the considered tag. They define a "data structure" in an object-oriented sense.

Since we store the meta information about data tables again in the form of sd-records we can use the same tools to retrieve and manipulate both, data *and* specifications. Even more importantly, such an approach allows flexible, modular, and independent extension and modification of the structure of the data base, like adding a new data table type for a different kind of application, since the meta information is a part of the data base, and not explicitly fixed in the tools of SymbolicData.

Tag attributes need to specify

- the type of the tag which determines the syntax of its value,
- a level of the tag which determines its importance (`level==1` characterizes mandatory tags),
- and a description of the meaning/purpose of the tag.

Further attributes may specify the name of a (Perl) procedure that semantically validates (e.g., verifying that polynomials are in normal form) or even generates (e.g., determines the number of variables occurring in a polynomial) the value of the tag, or defines how the tag value has to be transformed into a different format (e.g., how polynomials are represented in HTML).

The **type concept** for tag values we have developed can loosely be described as follows:

1. It defines (mostly by means of regular expressions) a set of basic tag types, e.g., `Text`, `Integer`, `Float`, `URL`, `Ref` (for references to records in other tables), `Polynomial`, `BibTeXEntry`, etc. These basic tag type specifications are again stored in form of meta sd-records which allows dynamic type extensions by simply adding a new sd-record specifying a new basic tag type.
2. It defines how lists and hashes can recursively be constructed from basic types. To have a list constructor is necessary to express such concept as "list of (lists of) polynomials". A hash constructor, which constructs sets of key/value pairs from the underlying type, is necessary to express, e.g., one-to-many or many-to-one relations between records and tags.

Requiring that each tag value is of a certain type has the advantage that many operations, like syntactic validation, HTML or SQL conversion, etc., on tag values can be realized in a generic, "content independent" way.

Interrelations between different tables are specified by means of the type `Ref`. A tag value of type `Ref` (or, reference, for short) is a hash of key/comment pairs where 'key' is the name of a record, or even a regular expression matching several records, in the foreign table and 'comment' is any text. The name of the foreign table is either specified in the tag's meta sd-file or inherited from the tag name, if it coincides with a valid table name. Interrelations are used to attach to a record, for example, bibliography entries (from the BIB table), problem descriptions (from the PROBLEMS table) etc.

Each meta table contains a special sd-record (whose Key is Meta) with "class attributes", i.e., information that specifies properties of the entire data table. This may be a description of the purpose of this data table, names of (Perl) modules required for processing records of this table, specifications of procedures which compare two records of this table, etc.

All sd-files are stored in a directory hierarchy, where the string concatenation of the Type and Key of a record yields the location of its sd-file within the directory hierarchy of the data base. A further sub-classification of the records of a table can be realized by means of the directory delimiter "/" in their Key values.

2.2 The SymbolicData Perl tools

The design of the SymbolicData tools has to take into consideration several circumstances. First, the operations they have to perform are of very different natures and requirements: they range from the insertion and validation of a single record, over the initiation, control and evaluation of benchmark computations on selected records, up to the transformation of parts or the entire data base into other representations like HTML or SQL. Second, the usability of these tools has to be as simple and as flexible as possible. And third, the tools need to be extendible at different levels.

With these circumstances in mind, the SymbolicData tools are designed to provide

1. a programming environment to be used for independent and rapid development of new components and specialized applications which, on the one hand, allows a maximum on code reusability and similarity of the look-and-feel of different components, and on the other hand, a maximum on flexibility and component independence.

2. a well-documented, flexible, and intuitive standard interface program which can initiate and control most of the implemented operations in a standardized and extendible way.

The SymbolicData Perl tools are the main vehicle for operations on the data base. They are implemented as a hierarchy of **Perl modules** which we divide into four categories:

Basic modules : They implement primitive operations, like I/O and tag/value access of sd-records.

Action modules : They implement the generic part of actions like validate, insert, compute, transform, etc. to be performed with the data base.

Table modules : They implement those parts of actions that are specific for a given table, e.g., how to validate a bibliography entry.

The symbolicdata program : It provides a standard interface that realizes command-line parsing, initialization of global variables and required modules, and execution of the well defined actions inherited from the command line.

To give the reader a feeling of how these modules cooperate we describe the main steps executed by the symbolicdata program. Its synopsis is

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

On start-up, symbolicdata loads all the basic modules, parses the command-line arguments up to the mandatory action argument(s), and loads the **global action hash** which specifies, in a well-defined format, 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 dynamically be extended at run-time using the first (optional) -req file argument, where file is the name of a Perl module which is loaded *before* the actions are parsed. Next, for each action, the modules listed in the respective action hash entry are loaded.

Then, symbolicdata initializes the **global command-line hash** which stores the recognized command-line options, their properties (like syntax of the argument, documentation, etc.) and (default) values. Each loaded module, including the basic modules, may add general, or action-specific entries to this global command-line hash. This way, the list of recognized command-line options is dynamically built up at run-time, and, hence, can independently be extended by other modules and is kept as small as possible. Values for command-line options can also be given in so-called init-files, which allow convenient editing and storing of these values.

After the modules are loaded and the command-line hash is set up, all remaining command-line arguments are parsed, and their values are stored in the appropriate slots of the command-line hash.

Finally, symbolicdata calls the specified action(s) in the order in which they are listed on the command-line: The first action gets the remaining command-line arguments as input, subsequent actions get the output of their preceding action as input, unless, of course, an error occurred.

The Perl tools use a hierarchy of hashes as **internal data representation** of the data base: the entire data base is a hash of Type/table pairs, a table is a hash of Key/record pairs etc. Furthermore, these hashes are implemented as so-called tied hashes, i.e., the basic hash operations like creation, value access, iteration, and destruction are overloaded. This overloading enables transparent data manipulations on both, the internal sd-record hashes and the external (persistent) sd-files. It also enables automatic loading, caching and storing of sd-records; read-only access of sd-records; automatic or explicit conversion of tag values into strings/lists/hashes, etc[1].

To increase the usability of the implemented tools, it is necessary to provide adequate and up-to-date **documentation** of their various features. From our experience, this is best realized by keeping the documentation and the source code closely together. Therefore, each module, action, and command-line option specification also has to provide well-defined hashes or hash entries which describe and illustrate the provided feature(s). This way, extensive documentation in various formats, e.g., a short ASCII description of relevant command-line options, or a detailed HTML table of all actions and their respective command-line options together with relevant examples, can be generated directly from the source code.

[1] Most of these features can be controlled by command-line arguments.

3 Realizing the Goals of SymbolicData

3.1 Collecting and maintaining data

To collect data from a certain application field one first has to specify the structure of the records to be collected. This requires to create one or several data tables via their meta tables.

As described above, a meta table consists of a set of tag descriptions, i.e., sd-files that can be created with any text editor and inserted at the right place via the `symbolicdata Insert` action. Each such meta sd-file contains the description of the attributes of a tag of the table to be defined.

Several such tag definitions (`ChangeLog`, `Comment`, `Date`, `PERSON`, `Version`) are predefined, i.e., inherited from a "master table" (which is an abstract class in object-oriented terminology). In particular, all records have a `PERSON` tag defined which is to be used as a reference to the table `PERSON` that collects information (affiliations, email addresses, etc.) of persons who contributed to SymbolicData. This guarantees a fair authorship management of different contributions along the GNU Public License conditions which applies to SymbolicData as a whole.

Furthermore, depending on the domain of the application, tag and/or table specific Perl functions might have to be implemented and specified in the meta sd-records which realize semantical operations like validation, generation, and comparison of tag values.

After the new table is specified, records of this table may be inserted into the data base. Each record has to be supplied as sd-file that either can be created by a text editor from a template or converted with appropriate Perl tools, possibly using the SymbolicData programming environment, from other formats.

New sd-records should be inserted into the data base using

```
symbolicdata Insert [options] sd-file(s)
```

This action first validates the given record, secondly, checks for uniqueness of the new record, and, thirdly, inserts the record as sd-file at the right place.

Validation first checks for correct flat XML syntax and presence and plausible values of all mandatory tags. Then, level by level, tag values are checked syntactically and, if a tag 'validate' and/or 'generate' function is defined in the corresponding meta sd-file, the tag value may also be semantically validated, or even generated.

After validation, the record is checked for uniqueness w.r.t. the existing records of the same table in the data base. This is either accomplished by a (semantical) 'compare' function defined in the table's meta sd-file or by the standard compare function that compares tag values by string comparisons modulo whitespaces. Note that a semantical comparison of two records may require certain elaborations since the same example may, e.g., occur with different variable names or in different representations.

In general, the evaluation of semantical aspects of records requires to cooperate with software capable of symbolic manipulations. For reasons of familiarity, personal preference, and suitability, we use, at the moment, only Singular [6] for such purposes. However, if it becomes necessary or convenient, other CAS could supplement or replace Singular as the underlying Computer Algebra engine.

3.2 Running benchmark computations

SymbolicData's `Compute` environment is set out to realize the following three goals:

1. To facilitate automated and trusted benchmark computations, that is, benchmark computations whose results w.r.t. time and correctness are repeatable, comparable, and trusted by the community.

2. To serve as a test-bed for developers, that is, as a tool with which developers of Computer Algebra software can conveniently and reliably evaluate new algorithms and implementation techniques.

3. To provide a repository of computational results which can be used for further development, like computing invariants of the original example, correctness verifications and timing comparisons of other computations, etc.

In this section, we present the main principles of the realization of these ambitious goals. See [9] for details, further explanations, examples and complete on-line documentation.

Analyzing the general nature of benchmark computations reveals dependencies on the following parameters[2]:

Example: The example which is to be computed, i.e., an sd-record which provides the object of the computation.

COMP: The actual computation to be performed, i.e., an sd-record of type `COMP` which describes the computation and serves as an interface to (Perl) routines, which examine an example for suitability for this computation, and, where applicable, check the syntactical and semantical correctness of the result of the computation.

CASCONFIG: A configuration of a Computer Algebra software which realizes the computation, i.e., an sd-record of type `CASCONFIG` which on the one hand, identifies the software, its version, and its implemented benchmark capabilities, and, on the other hand, serves as an interface to (Perl) routines which generate the input file and shell command to run the computation, which check the output of the computation for runtime errors, like out of memory, segmentation violations, syntax errors, and, if necessary, which perform (syntactic) transformations on the result such that it is suitable for further processing independent of the examined Computer Algebra software.

MACHINE: A description of the computer used for the computation. Such an sd-record of type `MACHINE` can automatically be generated by means of the action `symbolicdata ThisMachine` and further be used to specify the executables of particular `CASCONFIG`s.

Dynamic parameters: This includes specifications of: intervals for the run-time of a computation; which error, resp. verification, checks should be performed on the result; what to do with the output of the computation.

The benchmark computations of SymbolicData are facilitated by the Perl module `Compute` and realized using

[2] Where possible and reasonable, we encapsulate these dependencies into tables.

```
symbolicdata Compute [options] sd-file(s)
```

Parameter specifications are given either by command-line options, or, often more suitably, by init-files. A benchmark run consists of the following stages:

1. Check of correctness and completeness of input parameters.
2. Set-up of the computation.
3. Run of the computation.
4. Evaluation of the computation.

The set-up and evaluation stage require communications between the `Compute` module and the Perl routines specified by the input `COMP` and `CASCONFIG` records. The given input and expected output of these external routines is well-defined and documented. To ease the addition of new computations and systems to the available benchmark computations, as much functionality is provided by first, the `Compute` module; second, the routines of the `COMP` record, and, third, by the routines of the `CASCONFIG` record. For example, the run-time error check specification of a `CASCONFIG` can be as simple as specifying a regular expression.

Based on the input file and shell command returned by the `CASCCONFIG` routines, the actual run of the computation itself is fully controlled by the routines of the `Compute` module. For reliability reasons, timings are measured externally based on the GNU `time` program. While the actual computation is running, the `symbolicdata` program "sleeps" until either the computation finished, or the maximal (user plus system) time allowed for a computation expired. In the latter case, the running computation is unconditionally interrupted (killed) such that a following evaluation of the computation recognizes a "maxtime violation". Furthermore, if a run of the computation took less than a minimal (user plus system) time required, the computation is repeated until the sum of the times of all runs exceeds the bound, and the reported time is then averaged. Notice that the measured computation times include the times a system needs for start-up, input parsing, and output of result. While one could argue that these operations do not really contribute to the time of the actual computations, we did not separate out these timings (at least for the time being) for the following reasons:

- Mechanisms which isolate the pure computation time and do not rely on a system's internal facilities to measure timings are cumbersome to implement and would very much complicate the control and set-up of benchmark computations.
- Time measurements for computations which are not dominated by the pure computation time are mostly meaningless since start-up is a constant and I/O usually a linear operation w.r.t. the size of the input and output data.

The information about a particular benchmark computation is collected into a record of the type `COMPREPORT` which stores all input parameters and results, i.e., error and verification status, timings, output, etc., of the computation. Where applicable and requested, records of the `COMPRESULT` table are used to collect system independent, verified, and "trusted" results of computations. These `COMPRESULT` records may be extracted from one or more `COMPREPORT`s and may be used for further verifications and computations of invariants.

Running automated benchmark computations may quickly produce voluminous amounts of output data[3]. Hence, we need mechanisms which effectively maintain and evaluate this data:

First, note that this is a classical data base application. We are in the process of developing tools to translate benchmark data to SQL and to store them in a classical data base. However, even as data base application, the management of benchmark data is still rather challenging since benchmark data combines records, software, machines, algorithms, implementations, etc. into a high dimensional "state space" which needs to be analyzed.

Second, note that only tools to analyze benchmark data are not enough. To effectively compare benchmark runs we need standardized and widely accepted concepts and methods to statistically evaluate this data under various aspects. The `EvalComputation` module provides a first solution attempt. Since a detailed discussion of the involved aspects would go beyond the scope (and frame) of this paper we refer to `www.SymbolicData.org/doc/EvalComputations/` for a starting point for further thoughts and discussions.

3.3 Accessing and transforming the data base

One of the main purposes of digital data collections is to flexibly access, select, combine, sort, manipulate, etc. data from the underlying data base by varying principles, and to present the output in various formats.

Since standard data base programs allow much more flexible navigations through the underlying data pool, SymbolicData provides an interface to SQL which allows to define, create, and generate different SQL tables derived from tables of the primary data base. In particular, all interrelation information contained in the primary data base may be extracted to SQL relation tables and stored in your favorite (SQL compliant) data base. This interface, solely ASCII based at the moment, is defined via attributes in meta sd-files.

For presentation of data we use HTML and standard browser techniques. An HTML interface is best suited to present and browse data, to create different views, and trigger search. Interrelations can conveniently be realized by HTML links. As for today, we offer a scratch implementation (see `www.SymbolicData.org/Data`). A more elaborated interface is under development.

4 Two Examples

To illustrate the design principles described above, we describe in this section, by means of two examples, how tables should be designed and used. That is, we present and explain the structure of the tables of the two application fields where we started to collect data.

4.1 INTPS – a collection of polynomial systems

As a first application we tried to specify a framework to unify the different benchmark collections of systems of polynomi-

[3] For example, running a Groebner basis benchmark on the appr. 500 polynomial systems and 10 `CASCONFIG`s we have collected/implemented so far, produces appr. 1GB of data, among it, 5000 `COMPREPORT`s!

als as, e.g., [1, 2, 4, 8, 11, 12]. 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 polynomial 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 and to define the corresponding table `INTPS` accordingly.

A system of polynomials in `INTPS` is defined through its basis, list of variables, and list of parameters. The tags `basis`, `vars`, and `parameters` correspond to these entries. They are the most important tags: `basis` and `vars` of `level==1`, hence, mandatory; `parameters` of `level==2` since for $R = \mathbf{Z}$ there are no parameters.

For uniformity reasons and to ease comparison, we require of a "valid" INTPS record, that its basis polynomials are stored in expanded 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. Based on SINGULAR, the (Perl) `INTPS::validate` routine defined in the `INTPS` table module validates, and, if requested, necessary, and possible, "fixes" these properties of an INTPS record.

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.

Relational information relates the polynomial systems to other tables. This might be a bibliography reference of the origin of the example, bibliography references of papers that considered the example, a problem description of where the example came from or how it was generated from certain parameters, etc. Since relational information relates two tables we have to declare one of them as foreign and to attach the information to the other table. For `INTPS`, we define optional tags `BIB` containing a reference to the original bibliography source described in the `BIB` table and `PROBLEMS` containing a reference to a problem description in the `PROBLEMS` table. For the bibliography references to papers that consider the given example we declare the `INTPS` table as foreign, i.e., we define a corresponding `INTPS` tag in the `BIB` table. The main reason for this decision is persistence in the sense that we do not need to change an INTPS record each time a new publication refers to it. For similar reasons, the bibliography reference of the origin is attached to the `INTPS` table, not to `BIB`. Note that it is not always as easy as here to make such a judicious decision.

For integrity reasons, we furthermore need to assure that there are no "equal" records in our collection of `INTPS` records. The first problem we face here, is to decide what we actually mean by "equality" of `INTPS` records. Possible definitions range from equality of the ideals generated by the basis polynomials up to string equality of the `basis` tag values. With benchmark computations in mind, we decided on the following definition: Let $F = (f_1, \ldots, f_n) \in R[x_1, \ldots, x_m]^n$, $G = (g_1, \ldots, g_n) \in R[y_1, \ldots, y_m]^n$ be n-tuples of polynomials. Then we define F to be equal to G iff there exist permutations $\pi \in S_m, \sigma \in S_n$ such that

$$f_i(y_{\pi(1)}, \ldots, y_{\pi(m)}) = g_{\sigma(i)}$$

for all $1 \leq i \leq n$.

Having this definition at hand, we still need effective methods to actually determine the equality of two `INTPS` records: a brute-force, trial-and-error method is certainly computationally infeasible, since already by now we have `INTPS` records with polynomials in more than 40 variables. For this purpose, the first author has developed and implemented within SINGULAR an algorithm which uses structural information of the polynomials to significantly cut-down the number of possible permutations. Tested with random permutations on about 500 examples from our collection, the implementation needs at most a minute or so to recover the input permutations and hence, to decide the equality of `INTPS` records in the above sense. Details of the algorithm and its implementation will be given in a forthcoming publication.

4.2 GEO – a collection of mechanized geometry theorem proofs

As a second application of our general framework we collected examples from mechanized geometry theorem proving scattered over several papers mainly of W.-T. Wu, D. Wang, and S.-C. Chou, but also from other sources. The corresponding `GEO` table contains about 250 records of examples, most of them considered in Chou's elaborated book [3].

The examples collected so far are related to the coordinate method as driving engine as described in [3]. The automated proofs may be classified as constructive (yielding rational expressions to be checked for zero equivalence) or equational (yielding a system of polynomials as premise and one or several polynomials as conclusion).

To distinguish between the different problem classes we defined a mandatory tag `prooftype` that must be one of several alternations defined in the `Syntax` attribute in the corresponding meta sd-file. Extending/modifying this entry modifies the set of valid proof types. Hence the table is open also for new or refined approaches.

According to the general theory, see, e.g., [3], for a geometry proof in the framework under consideration one has to fix

- lists of independent (tag `parameters`) and, for equational proof type, dependent (tag `vars`) variables,
- formulas for the coordinates (tag `coordinates`) of all intermediate points, lines etc.,
- for equational problems, the polynomial conditions defining the relations between the dependent variables (tag `polynomials`),
- the conclusion polynomial(s) (tag `conclusion`),
- and possibly polynomial inequalities (tag `constraints`) which are required to be satisfied since the conclusion is invalid in general.

Further, we collect some background information of relational type and, for equational problems, also a "proof" (tag `solution`)[4].

[4] For constructive problems, a normal form computation of the rational expression obtained from the conclusion proves or disproves the theorem.

At the moment the background information consists of a reference to PROBLEMS as foreign table which points to a statement of the geometry theorem and, for equational type, a reference to the corresponding polynomial system in the INTPS table. References to bibliography entries are handled as above, i.e., GEO is considered as foreign table and the references are attached to BIB records.

We follow the spirit of [3] and collect not only the corresponding polynomial systems but also the way they are created from the underlying geometric configuration, i.e., the corresponding code of a suitable geometry software. To study aspects of code reusability and generality we took the GEOMETRY package [5] of the second author as base, that meanwhile exists in versions for REDUCE, MAPLE, MATHEMATICA, and MuPAD.

Due to different restrictions (case sensitivity, principal syntax differences), the code which describes a geometric statement in the GEOMETRY package language (Geo code, for short) varies between different CAS, but in a way that can be handled automatically. The tag values of coordinates, polynomials etc. contain code in a generic language that can be processed by Perl tools to generate correct Geo code for the different CAS. The design of this generic language may serve as a prototype also for other tables that store CAS code. We will not embark into details here, since this part works well for the special application but is yet under development.

The solution tag value contains code that is generic in a more obvious way. In most cases it contains the lines

```
sol:=geo_solve(polys,vars);
geo_eval(con,sol);
```

or

```
gb:=geo_gbasis(polys,vars);
geo_normalf(con,gb,vars);
```

where polys, vars, and con are assumed to be CAS variables that contain the polynomial conditions, variables, and conclusion and geo_solve, geo_eval, etc., are appropriate procedures for solving, evaluation, Groebner basis and normal form computation, that are defined in special interface packages, one for each CAS, in terms of the respective functionality of the given CAS. To really prove one of the given geometry theorems, the respective CAS must load the interface package as init-file and the SYMBOLICDATA tools must translate the given tag value into syntactically correct input for the given CAS.

5 The Current State of the Project

The SYMBOLICDATA project evolved as a permanent interplay between its two facets: collecting data and extending/improving concepts, design, and tools.

As of today, the SYMBOLICDATA contributors collected more than 1100 sd-records, wrote 40 Perl modules with more than 15 000 lines of code, and implemented 22 actions for the standard interface program symbolicdata. The following short alphabetical overview of tables which currently exist may give the reader a feeling about the overall structure of the data that was collected so far.

- **Table BIB**: Table for bibliography entries.

  Collects bibliographical information in BibTeX format, short abstracts, and relational information to the GEO, INTPS, and PROBLEMS tables.

- **Table CAS**: Table for general descriptions of Computer Algebra software.

  Collects information about the address, author, email, url etc. of the software, and also a short description.

- **Table CASCONFIG**: Table for configurations of Computer Algebra software to execute benchmarks, see section 3.2.

- **Table COMP**: Table for descriptions of computations, see section 3.2.

- **Table COMPREPORT**: Table for reports of executed benchmark computations, see section 3.2.

- **Table COMPRESULTS**: Table for the output of executed benchmark computations, see section 3.2.

- **Table GEO**: A collection of problems arising from mechanized geometry theorem proving, see section 4.2.

- **Table INTPS**: A collection of polynomial systems with integer coefficients, see section 4.1.

- **Table MACHINE**: Table of computers on which benchmark computations are performed, see section 3.2.

- **Table PERSON**: Table of developers/contributors who are involved with SYMBOLICDATA.

- **Table PROBLEMS**: More detailed background information and comments about different problems.

  This may be a problem description, a pointer to the origin of the problem, related CAS code, and/or certain key words.

We started first benchmark computations on Groebner bases, using various coefficient domains and monomial orderings. These benchmarks have been (and are) run on the more than 500 INTPS records using 10 versions of different Computer Algebra systems. Other benchmark computations on polynomial systems (like "solving", real root isolation, syzygy/resolution computations) are in preparation.

www.SymbolicData.org will soon become the central site of the SYMBOLICDATA project, containing its WWW-pages, and its CVS and FTP repositories. It will be related to the MEDICIS project [7] that "can be used by anybody to solve scientific calculations with the tools of computer algebra and symbolic computation. It can, in effect, put at your disposal hardware resources, software and expertise." (from their web pages).

6 Concluding Remarks

SYMBOLICDATA grew out of a "grass root initiative" of a small number of people. We think that this is the most natural and productive way to start up and realize such a project. During the development we have striven for a good balance between far-reaching ideas and usable, deliverable results. Most of the concepts and tools described in this paper have undergone major revisions, as we gained further experience with the subject. We thank all the developers of SYMBOLICDATA for their skill, patience, and vigor during our collaboration, and present this paper on behalf of this community.

SymbolicData has now reached a stage where its main concepts and tools are reasonably stable, general and approved. In other words, SymbolicData is ready to be shared with a greater community for use, further development, and extension. For this, we seek cooperations for the design and implementation of data collections from other areas of Computer Algebra.

Acknowledgments

We would like to thank the UMS Medicis and its staff (especially J. Marchand) for providing the hardware and software to set up `www.SymbolicData.org` and for letting us use their excellent computing facilities.

We also would like to thank the Fachgruppe Computeralgebra of the Deutsche Mathematiker Vereinigung, and especially G.-M. Greuel and H.-M. Moeller, for their valuable input and recommendations during the development of SymbolicData and for sponsoring the `www.SymbolicData.org` domain.

References

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

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

[3] Chou, S.-C. *Mechanical geometry theorem proving.* Reidel, Dortrecht, 1988.

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

[5] Gräbe, H.-G. Geometry - a small package for mechanized plane geometry manipulations, 1998. See `www.informatik.uni-leipzig.de/~compalg/software`.

[6] Greuel, G.-M., Pfister, G., and Schönemann, H. Singular version 1.2 User Manual . In *Reports On Computer Algebra*, no. 21. Centre for Computer Algebra, University of Kaiserslautern, June 1998. `www.mathematik.uni-kl.de/~zca/Singular`.

[7] The Medicis project, 1998. See `www.medicis.polytechnique.fr`.

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

[9] The SymbolicData project, 2000. Soonly available at `www.SymbolicData.org`. For the moment consult `www.informatik.uni-leipzig.de/~graebe/SymbolicData`.

[10] von zur Gathen, J. A factorization challenge. *SIGSAM Bulletin 26*, 2 (1992), 22–24.

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

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

[13] Wester, M., Ed. *Computer Algebra Systems: A Practical Guide.* Wiley, Chichester, 1999.

[14] Zimmermann, P., Bernardin, L., and Monagan, M. Polynomial factorization challenges, 1996. Poster at ISSAC-96, see also `www.inf.ethz.ch/personal/bernardi`.