A Tool for Packaging and Exchanging Simulation Results *

Dragan Savić
Faculty of Electrical Engineering
Ljubljana (SI)
Dragan.Savic@fe.uni-lj.si

Matevž Pustišek
Faculty of Electrical Engineering
Ljubljana (SI)
Matevz.Pustisek@fe.uni-lj.si

Francesco Potortì
ISTI-CNR
Pisa (IT)
Potorti@isti.cnr.it

ABSTRACT

Storing and exchanging simulation data is a common task among simulation practitioners, but quite often it becomes a challenge as huge quantities of data are not uncommon, and conversion between different formats can become an unwieldy task. After examining some of the needs of the telecommunications simulation community, we describe the architecture of the working prototype of a general purpose archiver and converter for big quantities of simulation data to be released as free software.

Categories and Subject Descriptors

E.5 [**Data**]: Files—*Organization/structure*; D.2.12 [**Software Engineering**]: Interoperability—*Data mapping*

General Terms

Management, Measurement, Standardisation

Keywords

Measurements, Simulation, Data, Archiving, HDF

1. INTRODUCTION

We describe the motivation, design issues and approach to the implementation of a low-overhead software package that can import simulation or measurement results into a common data structure, launch post-processing applications on the imported data, and store data or export it into various output formats.

Modern simulation packages and statistical tools use different, even proprietary formats for storing results. This can be a major obstacle for an efficient exchange of scientific data. Moreover, if we consider issues like storage and data management, documentation and meta description, analysis and display or data filtering, the need for a common format or a tool to handle simulation results becomes apparent. This topic has been addressed [1] in the framework of the European COST 285 [1] Action "Modeling and Simulation Tools for Research in Emerging Multi-service Telecommunications", a forum where researchers from all around Europe periodically meet to address issues related to the simulation of communications systems. The point made was that apparently no general purpose tools exist for exchanging big quantities of simulation data coming from different sources in different formats. Not only was the need for a common format for exchanging data highlighted, but also the need of feeding this data to different tools for post-processing, each requiring a different input format.

To better understand the scope of different requirements we define a reference model that encompasses the creation, flow and processing of data in the analysis of telecommunications systems. The main functional parts comprising such a model (simulators, data collectors, graphing tools, statistical tools) are covered by the many existing tools that are used by the research community; we focus on input data in form of simulation results (e.g., ns-2 [6] traces), or measurements (e.g., Tcpdump traces). Raw data can be post-processed (e.g., packet delay) and the results stored separately from the raw data or complementing them, so that further analysis is possible based on both raw and post-processed data. Finally, the results can be exported in various widespread output formats, like tabular ASCII data or XML.

Data storage is based on the HDF5 [3] data format, which was selected after the analysis of the available options. HDF5 has been successfully applied in several scientific projects; it enables efficient data storage and lookup. Among the features most relevant to our purpose, it provides support for extremely large quantities of data, meta descriptors, and embedded compression. A set of programming libraries is available in C and Python as well as in numerous others languages, which simplifies software development.

We propose a tool called CostGlue, which has a modular software architecture. That makes it possible to include future development contributions from other research communities. The core is a Python application called CoreGlue,

*This work was supported by European Commission under the COST 285 Action, by the CNR/MIUR program "Legge 449/97 (project IS-Manet), and by the Ministry of Higher Education, Science and Technology of the Republic of Slovenia (program P2-0246).

ValueTools'06 , October 14, 2006, Pisa, Italy

[1]COST stands for *Cooperation in the field of Scientific and Technical research*, see `http://www.cost285.itu.edu.tr/` for information on COST 285.

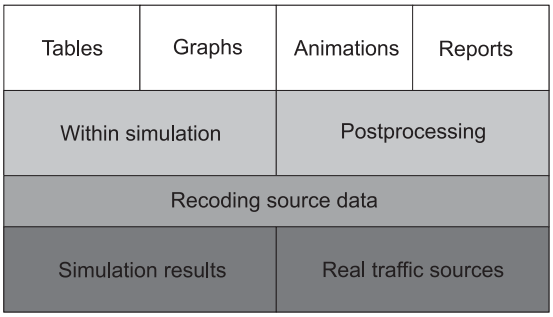

Figure 1: Reference model of a simulation process.

which provides HDF5 database connection for writing to and reading from the database. Specific functions, such as conversion from specific data formats and different calculations, are implemented as sets of self-descriptive loadable modules. A graphical user interface is envisaged via the use of a web browser, allowing user-friendly and efficient remote access to the application.

CostGlue will be released as free software. The next sections describe the proposed architecture of CostGlue into deeper detail, particularly its core - CoreGlue.

2. A MODEL OF THE SIMULATION PROCESS

In order to obtain a general and systematic overview of data creation, flow and processing, we define a reference model of the simulation process, which is depicted in Fig: 1. The model provides a layered decomposition of main functions that are usually encountered when using simulation as a research method. There are three layers in the model.

The first layer - *source layer* - provides raw simulation output, describing a simulation run into the smallest detail. Usually one or more records for every event during the simulation run are created at the source layer. Structure and format of the data at this point are entirely dependent on the simulator. Most frequently they are in the form of large tabular traces, in ASCII or binary format. An optional *source recoding sublayer* handles the raw source data: its main purpose is to convert between different formats (e.g., from ASCII to binary or vice versa), data compression (e.g., Gzip, Bzip2) or removing private information, e.g., header lines, from simulation traces. The source layer supports both raw simulation data and real measurements data. In fact, during our discussions, we found out that nearly the same model can be applied to the analysis of real network traffic traces. In this case the raw data are not a result of simulation, but rather, for example, data traces captured in a network link. Apart from the different tool (traffic capture tool, like Tcpdump or Ethereal instead of a simulator) that generates raw data, all the functions of the upper layers remain the same in both cases.

The *processing layer* is responsible for simulation data analysis. At this level cumulative results can be derived from raw data (e.g., mean packet delay is calculated) or statistical confidence of the results can be determined (and consequently additional simulation runs can be conducted). An important characteristic of the data processing layer is that the amount of data received from the source layer is usually much larger than the amount of results of post-processing.

The *presentation layer* is the final stage where the results are organized in a form useful for communicating the most important findings with other interested practitioners. In the case of simple tabular printouts of the results, this layer is void or only does trivial modifications of the data (e.g., changes in number formats, column spacing). However, data is frequently shown in 2- or 3-dimensional graphs (e.g., being part of scientific reports or research papers, web pages) or even presented in animated form (e.g., ns-2 NAM - Network AniMator). At this layer the predominant requirement is flexibility of presentation and a possibility to create new or modified presentation objects from new or changed simulation results without reformatting.

We can map the functionality of particular tools used in simulation to the layers of our model. Usually, a single tool provides more than one functional layer or even all of them. In the most favorable situation it would encompass all the functions needed and implement them adequately to meet all the researcher's needs. In practice this occurs very rarely and there is usually a set of complementary tools that covers the required scope of functions within the model. The selection of tools is made on arbitrary conditions, including their capabilities and performance, the researcher's past experience with a particular tool or the availability of tools. In the case of simulated results, raw data can be generated by discrete event simulators (e.g., ns-2, Opnet). Raw data can be captured in real networks, with sniffers, such as Tcpdump or Ethereal. Source recoding can be done with small dedicated tools (e.g., Gzip, Tcpdpriv) or different proprietary shell scripts. Data analysis can be performed with generic tools for mathematical computation (e.g., Octave [2], Matlab, Mathematica, Excel), special statistical tools (SPSS, R), or proprietary and dedicated programs or scripts. Often the simulation package provides the functionality for data processing and analysis and it is up to the researcher to decide whether this is adequate or an additional more powerful and flexible tool should be used. Besides dedicated graphing or animation tools, presentation ability can be provided in generic mathematical tools and sometimes even simulators.

3. FORMATS FOR EXCHANGING DATA

CostGlue acts as a central repository for data generated by various different simulation programs and as a converter from several output formats to several input formats. Therefore, it is important to know which programs and formats are generally used by telecommunications systems practitioners. To this end we used the information obtained from the COST 285 participants, representatives of more than ten European nations, about the kind of tools they use for their simulation work. We learned that no single simulation tool has a dominant position, but that there is a great variety of tools in use.

A specific questionnaire directed to the above mentioned people yielded some interesting results:

- apart from tabular data, other types of data organization, such as hierarchical or other elaborate structural

formats are rarely used,

- most of the time, data are used for statistics or graphing, other uses such as data mining are rare,
- the most common method for evaluating the statistical accuracy of simulation data is to use independent replications with the combination of different simulation times to assure long enough stationary intervals; equivalent correlation length (single simulation run) is rarely used; these methods are generally applied on a case-by-case basis, with the help of custom Bash, Perl or Python scripts,
- a single simulation run produces anywhere from 1 MB to 2 GB of data, and a simulation campaign requires from 1 to 100 runs, in the responses we got; measurements campaigns (as opposed to simulation) required from 1 to 5 runs, each generating from 100 MB to 50 GB of data,
- storage required varies from up to 1 GB for short-term storage to anywhere from 10 MB to 10 GB and more for long-term storage,
- used metadata include type, date, parameter values and their description, version tracking, configurations, simulation scripts, location,
- metadata are stored in different locations: coded into directory and file names, in separate files, in different storage location (e.g., under root directory, shared directories) inside files, in databases.

Among those who use a predefined output format, the most common appears to be the network simulator ns-2. Among the generic tools for mathematical computation and running simulations, Matlab appears to be used by many. A large part of the simulators is composed by standard scripting or programming languages and, in general, by ad hoc simulators. Concerning the tools used for post-processing or graphing, there is an even greater variety. These observations, while limited in scope, show that some sort of tabular ASCII format is of common use, and thus that being able to read and write ASCII tabular data is certainly a requisite for our proposed archiver and converter. Nevertheless, the variety of tools used also calls for a general way of reading and writing many formats: that is why we consider the modular architecture of CostGlue a necessary feature for the tool to be useful at all.

Another interesting point is that simulation data and measurement data have a lot in common, and a tool useful for one can be also useful for the other. However, measurements like simulation results are often output in particular formats, and an input converter is very frequently needed. An interesting feature that can be made part of CoreGlue is the ability to give a similar treatment to data coming either from measurement or from simulation of the same environment, and archive them in the same format. This is the reason why the first prototype of the CostGlue will include the ability to read ns-2 data and Tcpdump data, store them into a common format and write in Nam, ns-2 and Tcpdump formats. This capability would make it easy to use the many tools available that are able to analyze and graph data obtained by both ns-2 and Tcpdump.

All the above discussion leads to a scenario where a simulation tool is run several times, each time producing tabular data, that is, data that can be conveniently stored into a two-dimensional structure having relatively few columns and a possibly large number of rows. What about data that cannot be naturally converted to a two-dimensional format? In this case, the inner structure of the archived data needs to be different. One of the challenges is to make the tool efficient in the most common case of collections of tabular data, but still be useful in the case of non-tabular data.

4. THE DATABASE

A common file format solves several problems of simulation data exchange. Therefore we made a thorough analysis of different data formats and their corresponding libraries for data manipulation. Among many, we have focused on the following set of data formats: HDF4, HDF5, netCDF, ODB, FITS and OpenDX. Beside these, we also considered using plain text formats, XML and SQL databases. The results of the analysis makes it clear that the HDF5 file format is the most suitable for this task, since it meets all the requirements of data organization e.g., separation of raw data and metadata, and different requirements of contemporary computer system architectures, such as managing big quantities of data, offering a general data model, supporting complex data structures, portability among different computer platforms, parallel data access and processing, diversity of physical file storage media, and sustained development and maintenance.

4.1 Database structure

Summarizing the above, most simulation data in the computer communications area are collections of tables of numeric data: each simulation run generates a table of data having few columns and a possible large number of rows. Each table is associated with certain parameters specific to a simulation run that generated it, and is uniquely identified by the values of those parameters. We are interested in defining a database structure that is able to efficiently accommodate this type of data. HDF5 meets these requirements.

HDF5 is data format with associated software library. The software library consists of two primary objects: *dataset* and *group*. A *dataset* represents a multidimensional array of data elements, which can hold different types of data. The data stored in datasets can be either homogeneous (only one data type used within a single dataset - *simple dataset*) or compound (different number of data types within one dataset - *compound dataset*). Since tabular data collected from certain simulators often contains data with different types (e.g., integer, float, char), we use compound datasets to accommodate the nature of simulation outputs. An HDF5 *group* is a structure containing zero or more HDF5 objects. By using two primary HDF5 objects, data can be organized hierarchically by means of a tree structure where an arbitrary number of HDF5 objects are derived from the main "root" group. Groups and datasets have a logical counterpart in directories and files in a hierarchical file system and, similarly to a file system, one can refer to an object in an HDF5

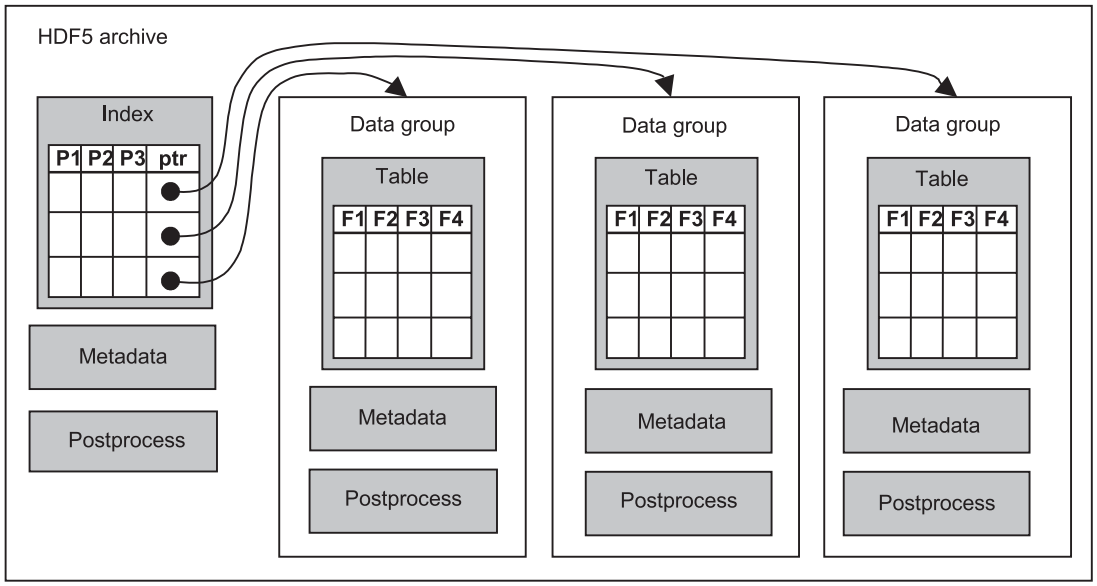

Figure 2: The logical structure of the database.

file by its full path name.

To meet the requirements of efficient data storage, especially those critical to management, understanding and reuse of scientific data, each HDF5 object may have associated *metadata* stored in the HDF5 file (referred as *archive*) in a simple attributes form. Attributes usually represent a small dataset connected to a certain group or a dataset. Their purpose is to describe the nature and/or the intended usage of the object they are attached to.

In the design of the database structure our goal was a flexible representation of the stored simulation data by using one multidimensional array, where a user can easily extract a desirable portion of the simulation data. Even though HDF5 supports multidimensional arrays, it is not efficient to store large amount of data in just one array. Furthermore, due to the HDF5 primary aspect of use, which is to have data organized hierarchically, storing everything inside a single multidimensional array would be a loss on the side of flexibility. We also introduced an indexing table which maps the logical view of a multidimensional matrix into an HDF5 hierarchical structure, as shown in Fig. 2.

Fig. 3 presents a detailed overview of the proposed database structure, where the indexing table represents the logical part and all other groups and datasets represent actual data, where the raw simulation data, metadata and *post-processing data* is stored. The whole database is treated as one archive containing a "root" group from which all other groups and datasets form a two-level tree.

An immediate extension to having tables of numeric data is having tables of fixed-length data, which can be flags, numbers or strings. The PyTables library allows efficient manipulation of 2-dimensional HDF5 compound datasets from Python referred to as *tables* from now on. Each table, together with metadata and post-processing data, is attached to a data group, which usually holds the data produced during a single simulation run. Data groups are indexed by vectors of parameters. An *index* is a 2-dimensional array, referred above as the indexing table, where the *parameters* relative to each data group are stored: each column corresponds to a different parameter, and each row contains the values of the parameter relative to a data group. Therefore an index is used as the data structure for accessing a data group, using an array of parameter values relative to that data group as the key. A table is attached to each data group, where each row is filled with the values of the *fields*, each field corresponding to a column of the table.

The overall structure is a collection of 2-dimensional tables indexed by arrays of P parameters. This can be logically seen as a matrix with P+2 dimensions, where the first P dimensions are sparse and the last 2 dimensions are dense. We define the first P indices as parameters identifying a data group. As for the last two indices, the first of them represents the field in the data group table, and the second index the row number of the data group table.

Let us now describe how the results of an example real-world simulation can be stored in the described structure. We are simulating the behavior of packet switches in ns-2; each run is characterized by several parameters, such as architecture type, buffer size, number of I/O ports and traffic load. Each simulation run differs in at least one of these parameters. When storing the results of one simulation run in the archive, we populate a new row in the indexing table: values in each row uniquely identify a simulation run. Each row includes the full path to a data group, containing

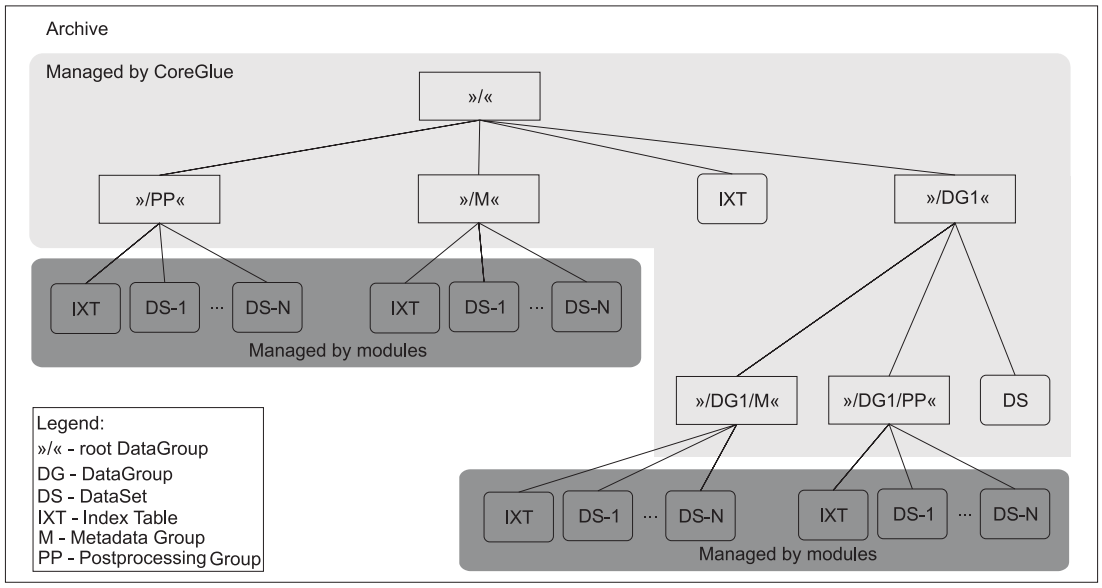

Figure 3: HDF5 database structure.

the table with the results of the simulation run, metadata and processing data, as shown in Fig. 3. Metadata stores information about the type of scripts used to generate that simulation run, the type of network topology, traffic patterns, etc. Post-processing data include packet loss probability, maximum, minimum and average packet delay. Metadata and processing data are also associated with the whole archive, and contain information relative to the whole set of simulation runs.

4.2 Metadata and processing data

No format is currently imposed on metadata and processing data; a simple possibility is to use a series of predefined `name=value` pairs, plus the ability of adding arbitrary data. This point will be the subject of future research. As an example of metadata in a related project, we mention Crawdad [5], an archive of wireless network trace data from many contributing locations, aimed at the research community. Here is an example of metadata used in Crawdad:

- Data
  1. Dataset - data set name and references
     (a) Traceset - trace set name and date
        i. Trace - file name
        ii. Trace - file name
     (b) Traceset - trace set name and date
        i. Trace - file name
- Tools - tools used for creating the traces
  1. Tool - name and download reference
  2. Tool - name and download reference
- Authors
  1. Author - name
  2. Author - name
  3. Author - name
- Papers
  1. Paper - title and references

5. COSTGLUE ARCHITECTURE

As described, the CoreGlue manages the index and the database structure, including the tables. Modules are responsible for metadata and post-process contents, both for the single data groups and for the whole archive. The CoreGlue and the modules together constitute the whole application, which is named CostGlue.

The CostGlue is written in Python. This language was chosen because of anecdotal evidence of its efficiency both in memory usage and processing power and for its programmability ease, due to automatic garbage collection and many native functions and types. Portability among operating systems is excellent and library availability for many tasks, especially mathematical ones, is rich. With respect to its main competitor, Java, Python has a generally smaller memory footprint and since it is completely free, does not suffer from being controlled by a single entity.

The architecture of the CostGlue application, depicted in Fig. 4, consists of a core, called CoreGlue, and several specialized modules. The core takes care of reading and writing to the HDF5 archive and of the dynamic loading of modules, while the modules are devoted to specialised tasks. Modules interact with the core through an API. Examples of tasks a module can perform are:

- data import/export to/from the archive,
- statistical computations over data stored in the archive,
- data extraction from the archive with complex filters,
- transformations over the data contained in the archive,

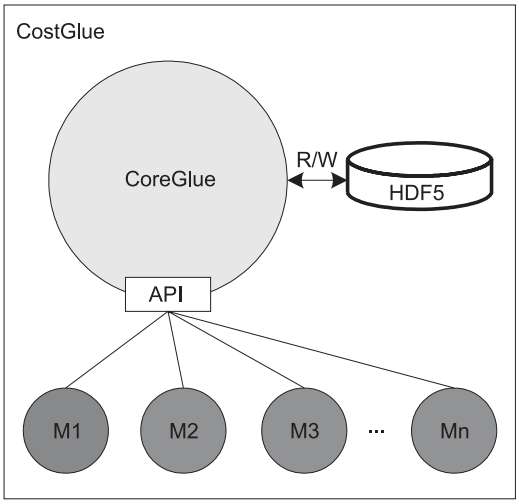

Figure 4: CostGlue architecture.

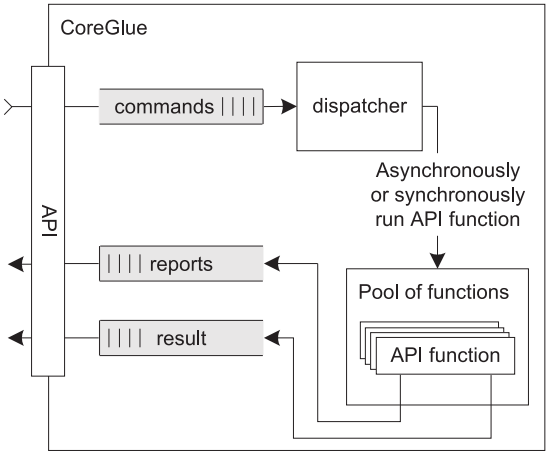

Figure 5: API and CoreGlue interaction.

- graphical output creating from the data in archive,
- provision of a generic graphical user interface (GUI) for exploring the data in the archive, doing simple import/export or computations and running other available modules, and
- provision of a module-specific GUI.

6. OVERVIEW OF THE MODULE API

Python, like many modern languages, supports dynamic module loading. We exploit this possibility by providing a module library with a limited number of modules, and leaving it open for other parties to write new modules. Modules are Python self-describing programs, which reside in a fixed place. The CoreGlue can look for available modules and query them one by one in order to get to know their capabilities; this can be used for instance for building a menu for a GUI (graphical user interface). Modules are able to describe the parameters they need and the type of output they produce. The needed parameters can then be provided to the module by the CoreGlue with or without input from the user; in the case where input is required, CoreGlue checks the provided parameters for consistency. A GUI can also use this information and present the choices to the user. Modules interact with the CoreGlue through a well-defined API which contains all necessary classes and methods for interacting with the database. The methods allow a module to manipulate the data group index in order to add or remove data groups, and to manipulate the data groups, in order to add or remove data, metadata and post-processed data.

The API also includes methods for accessing data with selectors written in index notation, which is widely used in matrix computations programs such as Matlab or Octave [4]. In this notation each of the P+2 indices can be "1", indicating the smallest index; "end", indicating the highest one; "n:m", indicating all the indices between n and m included; ":", indicating the whole range from smallest to highest, "n:s:m", indicating the range from n to m in steps of s; an array like "[1 5 6 8]" indicating the selected indices. Using the Matlab index notation one can take complex orthogonal slices of the multidimensional matrix composed of all the data in the database; in fact, the database structure can be seen as a sparse matrix with P+2 dimensions, and being able to take a slice of this matrix could prove a powerful feature of CostGlue.

6.1 The command line and the HTML GUI modules

When CoreGlue is invoked by a command line, its first argument is the module name, followed by the parameters needed by the module. When invoked with the argument `html`, the CoreGlue acts as an HTTP server, providing a graphical user interface. Through this interface, the user can look at the list of available modules and, for each of them, look at a description, at the input they require and at the output they provide. The CoreGlue provides an interface for the input of module parameters, complete with checking, thanks to the information that it reads from the modules themselves. In the simplest case, this is analogous to calling the module on the CoreGlue command line, but more convenient for interactive use. A module can also provide a graphical interface or graphical output by itself.

Currently an interactive command line module is implemented. This module is used for debugging of the CoreGlue and modules. Additionally, it can be used to import and export tabular data. This requires the user to specify options and arguments; a specialized module can automatically recognize the type of a trace and automatically provide naming, and possibly, filtering.

As shown in Fig. 5, the CoreGlue contains three different queues: *command*, *result* and *report* queue, which are used for exchanging different types of messages with modules. When using the CostGlue the first step is to call the CoreGlue together with a module name and its optional arguments. A given module can work either alone, e.g., by performing a batch job, or it can depend on other modules, e.g., a command line interface can call a specialized module to do some work. Modules can also differ in the way they are built. Simple ones, after sending the command to the

```
Group open (paramvals, exists, NOWAIT)
  message.name = 'OpenGroup';
  message.args = (paramvals, exists);
  retval = enqueue(message, CommandQ);
  if (NOWAIT)
    return retval;
  else
    assert(retval != error)
    dequeue(message, ResultQ);
    assert(message.name == 'OpenGroup')
    return message.result;
  endif
endfun
```

Figure 6: API command stub inside the CoreGLue.

CoreGlue, wait for the results and are therefore not able to perform any additional tasks. Alternatively, complex modules run asynchronously with respect to the core, so they can check periodically the result queue for results and in the meantime inform the user about the current work progress by reading report messages from the report queue. Modules working as described are run as separate threads inside the core.

6.2 API examples

All API commands are asynchronous, and return immediately if the `NOWAIT` option is given. However, modules can use the API in a more straightforward fashion, by not using the `NOWAIT` option; in this case, they will not be able to show any progress report, and will not be able to suspend, resume and stop a running API command. When an API command is called, the core creates a command message and enqueues it to the command queue from where it is read and executed by the *dispatcher*, which is a loop running as a separate thread. Fig. 6 sketches the implementation of the API command stub inside the core.

Here are some examples of commands provided by the API:

- `Archive open (filename, flags)` - opens an archive on disk: arguments are the same as in the C stdio library, returns an archive object,
- `Group open (paramvals, exists)` - `paramvals` is a list of parameter values; if `exists` is true, return the existing data group with the given parameters, else return a newly created group with the given parameters,
- `FieldList add (fieldlist)` - adds fields (that is, columns) to a table by taking a list of field names, types (e.g. Int8, Float32) and sizes of the fields.

Figures 7 and 8 are snippets of example code inside a module that makes several API command calls.

```
open an archive for writing
create a new data group
insert parameter values in index table
create a new table with given fields
open input file for reading
read the data into the table
exit
```

Figure 7: Example for data import.

```
open an archive for reading
go through index table
find the specified parameter values
get the data group name
get the table object
use selector to filter table data
create a new output file for writing
write the filtered data to the file
exit
```

Figure 8: Example for data export.

7. CONCLUSIONS

The purpose of the conversion and storage tool described in this paper is to facilitate the exchange and management of simulation data among researchers, and to ease the task of using different simulation, measurement, data processing and visualization tools, all having different input and output data formats. This design is under active implementation, and a prototype with a debugging interface is already working. As soon as the prototype passes its testing stage, it will hopefully be enhanced by other simulation practitioners thanks to its modular structure. Possible extensions to the presented architecture include being able to accommodate non-tabular data and defining useful structures for metadata and processing data.

We believe that software developed as part of research activity should be released with a free software license, because research results should be made available for use by anyone, for any purpose and be freely modifiable, in order to further knowledge and usefulness [7]. The choice of license will be among the MIT X license, the GNU LGPL and the GNU GPL licenses, which we think best serve the purpose of free research software.

8. REFERENCES

[1] Bragg, A. Observations and thoughts on the possible approaches for addressing the tasks specified in the COST 285 work-plan. COST 285 temporary document TD/285/03/15, CNUCE-CNR (IT), 2004.

[2] Eaton, J. W., and Rawlings, J. B. Octave – recent developments and plans for the future. In *proceedings of the 3rd International Workshop on Distributed Statistical Computing* (Mar. 2003), K. Hornik and F. Leisch, Eds.

[3] Folk, M., McGrath, R., and Yeager, N. HDF: an update and future directions. In *International*

Geoscience and Remote Sensing Symposium (IGARSS'99) (1999), IEEE, Ed., vol. 1, pp. 273–275.

[4] Golub, and Loan, V. *Matrix Computations*, 3 ed. The Johns Hopkins University Press, 1996.

[5] Kotz, D., and Henderson, T. CRAWDAD: A Community Resource for Archiving Wireless Data at Dartmouth. *IEEE Pervasive Computing 4*, 4 (Oct. 2005), 12–14.

[6] McCanne, S., and Floyd, S. *The network simulator - ns-2*. University of Berkley, Oct. 2005.

[7] Potortì, F. Free software and research. In *proceedings of the International Conference on Open Source Systems (OSS)* (July 2005), M. Scotto and G. Succi, Eds., ECIG Edizioni Culturali Internazionali Genova, pp. 270–271. Short paper.