Proposal for a Standard Time Series File Format in HDF5

A. Pfeiffer[1], I. Bausch-Gall[2], M. Otter[1]
[1]DLR Institute of System Dynamics and Control, Oberpfaffenhofen, Germany
[2]BAUSCH-GALL GmbH, Munich, Germany
Andreas.Pfeiffer@dlr.de, Ingrid.Bausch-Gall@bausch-gall.de, Martin.Otter@dlr.de

Abstract

This paper describes a proposal for a standard to store the results of dynamic systems simulations in form of time series data persistently on file. The reasons to develop such a standard are explained, as well as the decision to use the HDF5 file format as a basis. The meta-information to be stored on file is mainly deduced from the Functional Mockup Interface standard. Two variants are analyzed: Storing the meta-data either with a set of tables or in a hierarchy. Usability and performance measurements are utilized for the selection.

Keywords: Simulation Results; File Format; Time Series; Standard; HDF5; MTSF, FMI

1 Introduction

Many simulation programs store their simulation results in an own specific file format. However, modelers have to utilize simulation results from different tools in different ways, e.g. plotting in company specific formats, comparing the data with results from another simulation program or computing FFTs (Fast Fourier Transforms). Since often one tool is not suited for all these tasks, users or tool vendors have to implement API functions to access the result data from other programs. This is time consuming and has to be adapted when the format changes. Every program stores different information. Some store only the results, other more information such as units and names of signals. Many programs provide an open export of ASCII or CSV files, which makes data access easy. However, information supplied in these formats is not complete, reading the files is inefficient and storing and retrieving large amounts of data is not practical.

These issues exist since decades for almost all simulators in many physical domains. Many simulators offer a more or less mighty environment for result evaluation. But this is not their main development goal. Scripting tools such as Matlab [M12], Scilab [TSC12] or Python [P12a] are better suited to automate plotting of results with fine control of the layout, to generate standardized result evaluation reports, to perform signal processing (e.g. FFT), to compare with measurements, to run Monte Carlo simulations, or to perform optimizations over many simulations etc. The basic problem is then how to connect a simulation with a scripting environment. With a standardized time series file format, the approach from Figure 1 simplifies the task a lot, since simulation environments could generate files in this format and scripting tools could read files in this format directly.

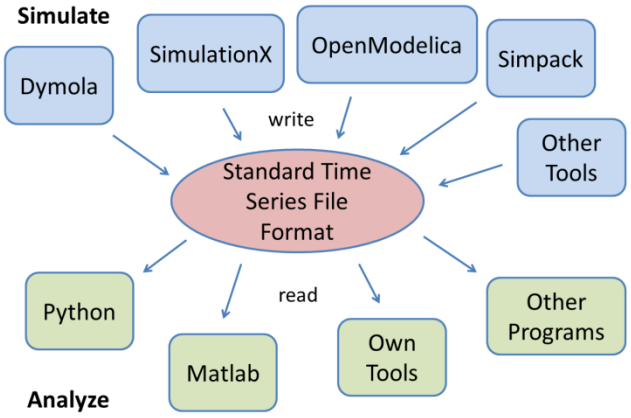

Figure 1: Standard time series file format and its interaction with tools.

In 2010, version 1.0 of the FMI (Functional Mockup Interface) standard was developed for the low level exchange of models and for co-simulation [MC10]. More than 30 tools support this standard already. Further progress can be achieved if these programs would support, at least optionally, the same result file format. For example, this would make it practical to automatically compare results of the same FMI model in different environments, and therefore the FMI import and export between tools could be tested in a much better way.

A standardized file format for simulation results would also be helpful for the Modelica community: More and more different Modelica simulators come to the market. Many components, models and libraries are developed in Modelica. They might be used in different simulators. It is necessary to compare results computed by different simulators automatically. In particular, the Modelica Association plans

to supply reference results for all simulation models available in the Modelica Standard Library [MA10]. This is only practical, if the Modelica tool vendors agree on a standardized result file format. Apart from testing, it might be desirable to collect results e.g. of all slaves in a co-simulation environment in one file.

1.1 Time Series Data

The basic purpose of the proposed file format is the efficient and compact storage of time series data, as shown in Table 1: The first column contains the values of the independent variable, usually time (but might be also another quantity, e.g. frequency), whose values must be monotonically increasing. A discontinuity occurs, if a value appears several times (here: at 0.4). Variable v is an example of a variable that depends on time t.

Table 1: Example for time series data.

Time t	Variable v	Variable w	…
0.0	2.8		
0.2	3.2		
0.4	5.1		
0.4	7.2		
0.5	6.9		
0.6	5.5		

If the variable is a continuous-time variable, then $v = v(t)$ is a continuous function and there exist also values of v between the tabulated points. Such intermediate points can be computed by interpolation of the tabulated values. If the variable is a discrete-time variable, then v is computed by a sampled data system at the values of the provided time instants. A value between the time points is not defined for v. If necessary, $v(t)$ with $t_j < t < t_{j+1}$ can be associated with the previous value $v(t_j)$ (= hold-semantics).

1.2 Name of the Standard

Results from the numerical integration of time dependent differential algebraic equations with discrete variable changes are typical *time series*. Since the standard shall be discussed, finalized and released by the *Modelica Association*, it is proposed to call it "Modelica Association Time Series File Format", shortly *MTSF*. This name is also used as the current extension of the corresponding files (e.g. robot.mtsf).

2 Selection of Basic Data Format

As a first step we collected requirements for such a file format and evaluated several existing formats against these requirements [BP11].

2.1 Requirements for the Result Format

A format for a time series file should fulfill the following requirements:

- Small and huge amounts of data (more than 10 GBytes) must be written fast and efficiently.
- Extraction of data from small and huge files must be fast.
- The format must be an internationally accepted standard.
- The standard has to be open.
- The format has to be also accepted by simulator developers outside of the Modelica community.
- It has to be future proof, which means stable support by the developers of the standard is expected and it has to be supported by many tools.
- The format should handle at least all data types of the FMI standard 1.0 [MC10] and the coming FMI standard 2.0 [MC12].
- It should be possible to add more data, if desired (e.g. diagrams of the model).
- APIs to standard programming languages like C, C++ and Fortran should exist.
- It should be easily accessible from scripting programs such as Matlab, Python, and others.

2.2 HDF5 Format

HDF, HDF4 and HDF5 (Hierarchical Data Format) [THG12a] are a set of file formats and libraries designed to store and organize large amounts of numerical data, originally developed at the NCSA (National Center for Supercomputing Applications at the University of Illinois). In 2005, the Hierarchical Data Format group was spinning off from NCSA as a non-profit corporation to ensure continued development of HDF technologies and the continued accessibility of data currently stored in HDF [NCS+12]. The HDF format, libraries and associated tools are available under a liberal BSD-like license. HDF is supported by many commercial and non-commercial software platforms, including Java, Matlab, IDL and Python.

The freely available HDF distribution consists of an API to access HDF files (implemented in C, with layers for C++, Fortran and Java), command line utilities, test suite sources, and the Java-based HDF Viewer to directly inspect HDF files. The currently existing two versions HDF4 and HDF5 differ signif-

DOI 10.3384/ecp12076495

icantly in design and API. The newer, more powerful HDF5 format consists of a hierarchy of objects where the leave objects are arrays. The dimensions of an array need not be known in advance and may be even constructed incrementally (as it naturally occurs in simulations). Many native data types are supported including all C data types. Furthermore, data can be compressed and graphics as well as videos can be stored. On the HDF web page applications with terabyte file sizes are reported (http://www.hdfgroup.org/why_hdf).

In [P10] a good overview of the features of HDF5 is given. It is suggested to use HDF5 to store simulation data. The reference highlights the following features of HDF5: The tree structure for convenient storage of data; HDF is a numerical aware middleware; the files and APIs allow portability, maintainability, compatibility of the user software; the openness of the software and the trustworthiness of the support.

2.3 Alternatives to HDF5

In order to handle efficiently large result data, only binary formats seem to be suitable. In principal also zipped xml-files might be applicable, but there seems to be still quite a large overhead to store and retrieve structured numerical data in such a format.

There exist also other open source binary file formats, in particular:

- **NETCDF**[1] from UCAR (University Corporation for Atmospheric Research). The latest version of NETCDF is a subset of HDF5 and the NETCDF files are therefore compatible to HDF5 (see "Format Descriptions" in http://en.wikipedia.org/wiki/NetCDF).
- **CDF**[2] from NASA. The CDF format is not compatible to HDF5. CDF seems to be also widely used and is, e.g. supported in Matlab and Python. CDF supports a set of arrays, but it does not support an object hierarchy. In this respect the HDF5 format is more powerful.

Another alternative could be to not base the design on a general purpose file format, but on a special binary format dedicated solely to time series data:

- Such a format could be newly designed and implemented. However, it would be a large effort to develop, implement and support an API that writes time series data in a subset of a HDF5-like data structure. Therefore we decided to not follow this approach.
- Another option would be to use one of the formats of ASAM (Association for Standardisation of Automation and Measuring Systems) [A12]. ASAM was founded in 1998 as an initiative of German car manufacturers with the goal of offering a platform for the development of universal standards such as MCD-2 MC, MDF, HIL V1.0.1 and ODS. A standard like ASAM MDF (Measurement Data Format) can be compared to the MTSF approach. It is designed to store and retrieve data from measurements. This standard is widely used in automotive industry. HDF5 and ASAM standards are, e.g., compared in [PA11]. There exists no open source API from ASAM to read and write data. The standard texts are available for ASAM members (with expensive membership fees for industrial partners) or can be bought for a pricey fee. For these reasons, ASAM standards seem to be not suited as general exchange format for time series result data between many tools.

Since all requirements of section 2.1 are fulfilled by the HDF5 format and there seems to be no equally suitable competitor, we decided to base the MTSF format on HDF5. Once the base file format is decided, the important question is what data to store? Our main target is to store simulation result data from tools that support the FMI standard [MC10, MC12]. Therefore, the time series data and associated meta-information to be stored is based on this standard.

3 Structure of the File Format

The basic structure of an MTSF file is shown by means of an example using screen shots from HDFView [THG12b]. The example file is based on the numerical integration of a Functional Mockup Unit (FMU) [MC10] by the open source simulator PySimulator [PHH+12]. The FMU was generated by Dymola [DS12] from the model `Modelica.Mechanics.Multibody.Examples.Systems.RobotR3.fullRobot` of the Modelica Standard Library [MA10]. The complete hierarchy of the result file is shown in Figure 2.

On the top level, the file shows two groups named *ModelDescription* and *Results*. *ModelDescription* contains the meta-information of the variables. The time series data of these variables is stored under *Results*. In order to read the result data of one or more variables, parts of the *ModelDescription* in-

[1] http://www.unidata.ucar.edu/software/netcdf
[2] http://cdf.gsfc.nasa.gov

formation has to be inquired in order to determine the location where the result data is stored.

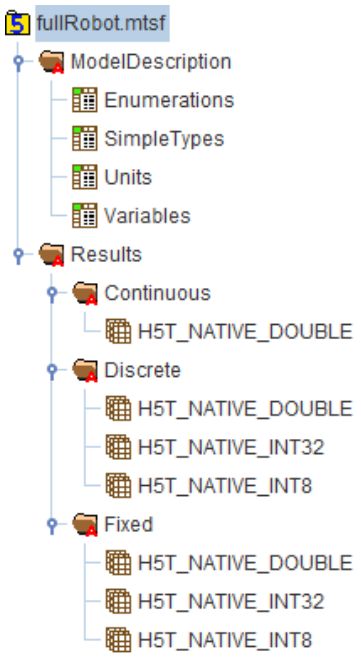

Figure 2: HDF5 hierarchy of the example result file.

The root directory has an attribute *mtsfVersion* that contains a string value for the version of the underlying MTSF format, see Figure 3. All groups and datasets from Figure 2 are described in the next subsections.

Name	Value
mtsfVersion	0.3

Figure 3: HDF5 attribute on root level of the result file.

3.1 Model Description

The HDF5 group *ModelDescription* contains a set of attributes (see Figure 4) which give (optional) information about the source of the model used for the experiment. The information is based on the coming FMI 2.0 definition [MC12].

Name	Value
author	
description	6 degree of freedom robot with path planning, controll
generationDateAndTime	2011-03-30T07:52:02Z
generationTool	Dymola Version 7.4 FD01, 2011-02-24
modelName	Modelica.Mechanics.MultiBody.Examples.Systems.Rc
variableNamingConvention	structured
version	3.2

Figure 4: HDF5 attributes of group *ModelDescription* in the example result file.

Variables

The HDF5 dataset *ModelDescription/Variables* (see Figure 6) defines the variables whose data is stored in the file. The HDF5 type definitions of the dataset *Variables* are displayed in Figure 5:

- *name* contains the names of the respective variables.
- *simpleTypeRow* defines the data type and the unit of the variable by providing the row index of the related simple type in dataset *ModelDescription/SimpleTypes* (see below). For example *simpleTypeRow* = 33 means that the type is defined in row 33 of *SimpleTypes* which means *Modelica.SIunits.Angle* (see Figure 8).
- *causality* and *variability* are HDF5 enumerations and provide information about the nature of the variable.
- *description* is a short description string of the variable.
- *objectId* and *column* provide the information where the data is stored for this variable (more details are given in section 3.2).
- *negated* is introduced to enable negated alias variables. It can only have the values 0 for *false* or 1 for *true*. The value 1 indicates that the values for this variable (stored in the data matrices under *Results*) have to be negated.

Name	Type
name	String, length = 56
simpleTypeRow	32-bit unsigned integer
causality	enum (input=2 local=4 option=5 output=3 parameter=1)
variability	enum (constant=1 continuous=5 discrete=4 fixed=2 tunable=3)
description	String, length = 144
objectId	Object reference
column	32-bit unsigned integer
negated	enum (false=0 true=1)

Figure 5: HDF5 variable types for the columns of the dataset *ModelDescription/Variables* in the example result file.

Instead of *objectId* and *column* it would also be possible to use an HDF5 region reference. This is a HDF5 link to the region of a data matrix, in our cases, e.g. a column of one of the matrices under *Results/Continuous*. A typical region reference looks like 0:3396963{ (0,685)-(599,685) } in HDFView where 3396963 is the HDF5 object id of the matrix. The region is selected by row 0 up to row 599 of column 685. The drawback of the region reference is that it is not supported to link to a whole column of a matrix. The row indices of the region have to be specified, too. Because the number of result points is generally not known before a simulation, the row indices of the region reference have to be updated at the end of the simulation process, which is quite an overhead if many variables are present.

DOI 10.3384/ecp12076495

	name	simpleTypeRow	causality	variability	description	objectId	column	negated
200	axis1.motor.D	0	1	2	Damping constant...	2831	2180	0
201	axis1.motor.J	43	1	2	Moment of inertia ...	2831	2179	0
202	axis1.motor.Jmotor.J	43	1	2	Moment of inertia	2831	2050	0
203	axis1.motor.Jmotor.a	35	4	5	Absolute angular ...	4013	153	0
204	axis1.motor.Jmotor.flange_a.phi	33	4	5	Absolute rotation ...	4013	325	0
205	axis1.motor.Jmotor.flange_a.tau	53	4	5	Cut torque in the fl...	4013	93	1
206	axis1.motor.Jmotor.flange_b.phi	33	4	5	Absolute rotation ...	4013	325	0
207	axis1.motor.Jmotor.flange_b.tau	53	4	5	Cut torque in the fl...	4013	503	1
208	axis1.motor.Jmotor.phi	33	4	5	Absolute rotation ...	4013	325	0

Figure 6: Dataset *Variables* in group *ModelDescription* of the example result file.

Simple Types

Name	Type
name	String, length = 54
dataType	enum (Boolean=3 Enumeration=5 Integer=2 Real=1 String=4)
quantity	String, length = 28
relativeQuantity	enum (false=0 true=1)
description	String, length = 1
unitOrEnumerationRow	32-bit integer

Figure 7: HDF5 variable types for the columns of the dataset *ModelDescription/SimpleTypes* in the example result file.

The dataset *SimpleTypes* (see Figure 8) contains a definition of the simple data types used in the variable description. The HDF5 data type definition of the columns is depicted in Figure 7. The simple data type can optionally have values for the string fields *name* and *quantity*. *dataType* is an HDF5 enumeration that specifies the basic data type (for example *Real* has the value 1). The default value for *unitOrEnumerationRow* is −1 (means no row) and for *relativeQuantity* it is 0. The *relativeQuantity* can only have values of 0 or 1 that represent *false* or *true* (this is only relevant if unit conversion takes place) If *dataType* is equal to 5 (= Enumeration) the value of *unitOrEnumerationRow* corresponds to a row in the dataset *ModelDescription/Enumerations*, otherwise to a row in the dataset *ModelDescription/Units*.

Units

Each simple data type can have a unit and several display units. Display units for one simple data type can be defined by using a row block in the dataset *Units*, see Figure 9 and Figure 10. A unit can have three different modes: base unit, display unit or default display unit. The value of *unitOrEnumerationRow* has to correspond to a row in *Units* with mode = 0 (BaseUnit), if there is a unit definition. If some display units apply for this base unit, they have to be listed in the rows below the base unit. Each mode of the display units can be 1 (DisplayUnit) or 2 (DefaultDisplayUnit). Only one display unit may have *mode* = 2. If no display unit has *mode* = 2, the base unit is used as default display unit. The base unit can only have *mode* = 0. If there is no unit (and no enumeration) for a simple data type, then the value for its *unitOrEnumerationRow* is equal to −1 (default value).

For example, the simple data type *Time* (see row 56 in Figure 8) is a Real data type with a unit that is defined in column 33 of the dataset *Units*. Here the base unit is *s* and the display units are defined by rows 34 up to 37 in the dataset *Units* (Figure 9). So, the display units are: *ms*, *min*, *h* and *d* with corre-

	name	dataType	quantity	relativeQuantity	description	unitOrEnumerationRow
28	Modelica.Mechanics.MultiBody...	5		0		0
29	Modelica.Mechanics.MultiBody...	1		0		-1
30	Modelica.Mechanics.MultiBody...	4		0		-1
31	Modelica.Mechanics.MultiBody...	1		0		-1
32	Modelica.Slunits.Acceleration	1	Accelerati...	0		24
33	Modelica.Slunits.Angle	1	Angle	0		28
34	Modelica.Slunits.Angle	1	Angle	0		30
35	Modelica.Slunits.AngularAccel...	1	AngularAc...	0		27
36	Modelica.Slunits.AngularVelocity	1	AngularVe...	0		26
37	Modelica.Slunits.Capacitance	1	Capacitan...	0		4
38	Modelica.Slunits.Current	1	ElectricCu...	0		2
39	Modelica.Slunits.Diameter	1	Length	0		22
40	Modelica.Slunits.Distance	1	Length	0		22
56	Time	1	Time	0		33

Figure 8: HDF5 dataset *ModelDescription/SimpleTypes* in the example result file.

sponding values for *factor* and *offset* in the style of FMI 2.0 [MC12]. The default display unit is *s*. This allows, e.g. a plotting program to display the results in different units by using the conversion factors stored in the *Units* group.

	name	factor	offset	mode
26	rad/s	1.0	0.0	0
27	rad/s2	1.0	0.0	0
28	rad	1.0	0.0	0
29	deg	57.295779...	0.0	2
30	rad	1.0	0.0	0
31	deg	57.295779...	0.0	1
32	s	1.0	0.0	0
33	s	1.0	0.0	0
34	ms	0.0010	0.0	1
35	min	60.0	0.0	1
36	h	3600.0	0.0	1
37	d	86400.0	0.0	1

Figure 9: HDF5 dataset *ModelDescription/Units* in the example result file.

Name	Type
name	String, length = 9
factor	64-bit floating-point
offset	64-bit floating-point
mode	enum (BaseUnit=0 DefaultDisplayUnit=2 DisplayUnit=1)

Figure 10: HDF5 variable types for the columns of the dataset *ModelDescription/Units* in the example result file.

Enumerations

The dataset *ModelDescription/Enumerations* (see and Figure 12) lists all enumerations that are defined in the model variables, i.e. variables of type Integer that can have only a small number of Integer values and a string is associated with every value. A plot program may then use the enumeration name instead of an integer to mark the value in an axis. The value of *unitOrEnumerationRow* corresponds to a row in the dataset *ModelDescription/Enumerations*, if the data type of a simple type is equal to 5 (= Enumeration). Enumerations do not have units, so there is no conflict with unit definitions.

Name	Type
name	String, length = 14
value	32-bit integer
description	String, length = 81
firstEntry	enum (false=0 true=1)

Figure 11: HDF5 variable types for the columns of the dataset *ModelDescription/Enumerations* in the example result file.

The row of *Enumerations* that corresponds to *unitOrEnumerationRow* has to have *firstEntry* = 1. The *firstEntry* column marks a new row block of enumerations. Each enumeration has a name and an integer value and may have a separate description string for example, the simple type *StateSelect* is an enumeration type with *unitOrEnumerationRow* = 7, it means in row 7 of *Enumerations* the defining enumeration block starts from "never" (1) up to "always" (5). Values for enumeration types are stored as integer.

	name	value	description	firstEntry
0	NoGravity	1	No gravity field	1
1	UniformGravity	2	Uniform gravity field	0
2	PointGravity	3	Point gravity field	0
3	NoInit	1	No initialization (sta...	1
4	SteadyState	2	Steady state initializ...	0
5	InitialState	3	Initialization with init...	0
6	InitialOutput	4	Initialization with init...	0
7	never	1		1
8	avoid	2		0
9	default	3		0
10	prefer	4		0
11	always	5		0

Figure 12: HDF5 dataset *ModelDescription/Enumerations* in the example result file.

3.2 Time Series Results

The numeric data associated with the defined variables is stored under *Results*. The HDF5 attributes of *Results* in Figure 13 include the most important parameters for the simulation experiment. *ResultType* defines the kind of the experiment, here: *Simulation*. The other attributes depend on the value of *ResultType*. For example, a result type *Measurement* has other attributes than a result type *Simulation*, but the attributes are standardized. The values of the attributes are optional with empty strings as default. Standardized attributes are necessary to exchange the attributes between different tools.

Name	Value
ResultType	Simulation
algorithm	BDF (IDA, DassI like)
author	Mr. X
cpuTime	
description	
generationDateAndTime	Tue, 17 Apr 2012 09:45:49
generationTool	Python
machine	Pluto
relativeTolerance	1.0E-4
startTime	0.0
stopTime	2.0

Figure 13: HDF5 attributes of group *Results* in the example result file.

The experiment may provide several time series under *Results*. Example names for the time series are *Continuous* for continuous-time variables, *Discrete* for discrete-time variables which change their values only at events, and *Fixed* for variables that do not depend on an independent variable (constants and

DOI 10.3384/ecp12076495

parameters). Additional groups might correspond to different clocks (e.g. a group for a periodic sample rate of 2 ms and a group for a periodic sample rate of 7 ms).

The group names of the time series can be freely chosen. Every time series (corresponding to a separate HDF5 group) may be associated with an independent variable. Therefore, each time series group has the attributes *independentVariableRow* and *interpolationMethod*. For example the attribute definitions of groups *Continuous* and *Discrete* are shown in Figure 14.

Continuous

Name	Value
independentVariableRow	0
interpolationMethod	linear

Discrete

Value
1
constant

Figure 14: HDF5 attributes of the groups *Results/Continuous* and *Results/Discrete* in the example result file.

The value of *independentVariableRow* is the row index in the dataset *ModelDescription/Variables* and defines the variable that is used as independent variable for the relevant data. In our example the independent variable of *Continuous* is variable *Time* that has a row index of 0. The independent variable of *Discrete* is variable *DiscreteTime* that has a row index of 1. For group *Fixed* the index *independentVariableRow* is equal to −1 in order to indicate that the variables are constant and do not depend on an independent variable.

The value of *interpolationMethod* is *linear, constant* or *clocked* and indicates how the numeric data values corresponding to the time series have to be interpreted. *Linear* means that piecewise linear interpolation is suggested between the given points. *Constant* means that the value of a variable for a point of time is held constant until the next point of time. *Clocked* means that no interpolation should be applied and only the values at the stored time points should be shown in a plot. Typically, *linear* is applied for continuous-time variables, *constant* for discrete-time variables that have an explicit value between event points, and *clocked* for sampled variables.

All time series data under a group like *Continuous* are stored in matrices. The column of such a matrix corresponds to one or more model variables and the row corresponds to the values of the independent variable. All elements of a matrix have the same HDF5 data type and the name of this data type is used as name of the matrix. In Figure 2, there are three matrices under *Discrete* of the types *H5T_NATIVE_DOUBLE*, *H5T_NATIVE_INT32*, and *H5T_NATIVE_INT8*. These are HDF5 data types and mean the matrices have a 64 bit floating type, a 32 bit integer type and an 8 bit integer type, respectively. In the latter matrix, the data of Boolean variables is stored as value 0 or 1. A basic Boolean type is not available in HDF5.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	0	0	0
2	0	0	-1	0	0	0
3	0	0	-1	0	0	0
4	0	0	-1	0	-1	0
5	0	0	-1	0	-1	0
6	0	0	-1	-1	-1	0
7	0	0	-1	-1	-1	0
8	0	1	-1	-1	-1	0
9	0	1	-1	-1	-1	0

Figure 16: Parts of the dataset *Results/Discrete/H5T_NATIVE_INT32* from the example result file.

Parts of the matrix *Results/Discrete/H5T_NATIVE_INT32* are shown in Figure 16. Each column of the matrix contains the numeric data of one or more discrete integer variables. The time values for the data are stored in a column of the matrix *Results/Discrete/H5T_NATIVE_DOUBLE* (see Figure 15). The column index is given in *column* of *ModelDescription/Variables* for the independent variable *DiscreteTime*. In the example file the column index

	0	1	2	3	4	5	6	7
9	0.004361142295306215	26.4	-10.7...	-15.0	5.39...	5.39...	6.87...	14.39...
10	0.0043611423912718415	26.4	-10.7...	-15.0	5.39...	5.39...	6.87...	14.39...
11	0.004667320075657221	26.4	-10.7...	-15.0	5.39...	5.39...	6.87...	14.39...
12	0.004667320085177664	26.4	-10.7...	-15.0	5.39...	5.39...	6.87...	14.39...
13	0.11742424242424244	0.0	-0.0	-0.0	0.0	0.0	0.0	0.0
14	1.238620759479847	-26.4	10.79...	15.0	-5.3...	-5.3...	-6.8...	-14.3...
15	1.3486863157736644	-26.4	10.79...	15.0	-5.3...	-5.3...	-6.8...	-14.3...
16	1.3497196258220059	-26.4	10.79...	15.0	-5.3...	-5.3...	-6.8...	-14.3...
17	1.3503821984874294	-26.4	10.79...	15.0	-5.3...	-5.3...	-6.8...	-14.3...

Figure 15: Parts of the dataset *Results/Discrete/H5T_NATIVE_DOUBLE* from the example result file.

is 0. So the first column of *Discrete/H5T_NATIVE_DOUBLE* represents the time for all discrete variables. All matrices of a time series group have the same number of rows: They are based on the same independent variable values.

4 Performance Tests with Python, Dymola and Matlab

We used Python 2.7 [Py12] to implement a test environment for writing and reading MTSF files. The Python(x,y) distribution (version 2.7.2.1) [P12b] includes the HDF5 interface h5py (version 2.0.1) [H12], which provides high level interface functions in Python for HDF5 files. In a second step, reading MTSF files by Matlab [M12] is tested.

4.1 Hierarchical Variables Concept

In the initial design phase of the MTSF format a different (alternative) format has been investigated than presented in Section 3. In this section we shortly explain this alternative format (called *hierarchical variables concept*), because it seems to be straightforward to save hierarchically structured variables in a HDF5 group hierarchy. However, the performance measurements in Section 4.2 and 4.4 indicate that the table-based approach of section 3 is better.

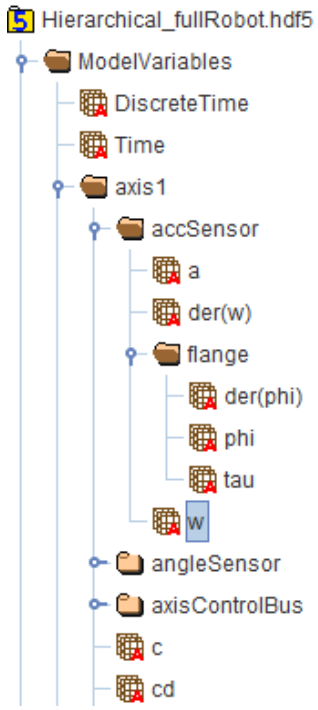

Figure 17: Parts of the HDF5 hierarchy using the hierarchical variables concept for the example result file.

The one to one mapping of hierarchical variable names to HDF5 groups and datasets is the main difference to the MTSF format presented in Section 3. For example, for the variable `axis1.accSensor.w` the dataset *ModelVariables/axis1/accSensor/w* in Figure 17 contains all the necessary information about the variable. The *ModelDescription* group (see Section 3.1) with its compound datasets is not present in this concept.

For first testing purposes the deepest dataset for each variable is only a 1x1 dataset containing an HDF5 object reference to one of the matrices under e.g. *Results/Continuous*. The numeric data for the variable is stored in one of the columns of the referenced matrix. The column index is stored as attribute to the reference dataset. It would also be possible to use an HDF5 region reference instead of the object reference and the column index. The resulting files sizes would only differ slightly.

Additional information (data type, unit, etc.) of each variable can be stored as further attributes. Some attribute examples are listed in Figure 18. To get the whole information included in the MTSF format of Section 3, much more attributes (or dimensions of the dataset) would be necessary. We have not worked it out so far.

Name	Value	Type
columnIndex	685	32-bit integer
dataType	1	enum (Boolean=3 ...
declaredType	Modelica.SIunits.AngularVelocity	String, length = 32
description	Absolute angular velocity of flange	String, length = 35
variability	continuous	String, length = 10

Figure 18: Some HDF5 attributes of each hierarchical variable dataset.

The advantage of the hierarchical variables concept is the hierarchical mapping of the model variable names and the HDF5 groups / datasets in the *ModelVariables* tree. Therefore, HDFView shows automatically a tree of the model variables when browsing through the groups. The main disadvantage of the hierarchical concept is the possibly large number of HDF5 objects building the *ModelVariables* tree. Intuitively, this is similar to a file system: Reading or writing 10 files (corresponds to the MTSF format of Section 3) is more efficient than reading or writing 10000 files (corresponds to the hierarchical variables concept) in which the same data is stored.

In the next subsection we compare the files that result from the two different concepts: hierarchical storage of variable information vs. the final MTSF

DOI 10.3384/ecp12076495

format of Section 3 with few HDF5 compound datasets in *ModelDescription*.

4.2 File Sizes

In MTSF files the HDF5 compression of objects with the gzip algorithm can be used which is very effective for the meta-information, whereas compression with gzip in the hierarchical variable concept is technically not possible. The reason is that the meta-information of one variable of the hierarchical concept is stored in an HDF5 group, and HDF5 does not support compression of such an object.

The binary result file of the Modelica modeling and simulation environment Dymola [DS12] is used as a reference to compare the file sizes generated in HDF5. This proprietary storage format of Dymola is very compact. Dymola stores variables with different names and same data (so called alias variables) just once. Dymola also stores negated alias variables, i.e. the numeric data of two variables `a = -b` only once. If `b` is stored, for variable `a` only the information is stored that it has the values of `-b`. This aliasing method is also used in the MTSF and in the hierarchical format.

As the performance measurements below indicate, storing many (more than 1000) objects in HDF5 with standard options is very storage consuming. The storage requirements can be considerably reduced by using the following two options [THG11]:

- For the storage strategy of objects the option *Compact* (in Python: `h5py.h5d.COMPACT`) should be used, instead of *Contiguous* or *Chunked*. This option leads to storing the raw data of small datasets in the header of the dataset.
- In HDF5 1.8.0 an optional mechanism is introduced to store groups much more efficiently by using a fractal heap and indexed with an improved B-tree. In order to activate this feature, the version number in which the HDF5 file is generated needs to be specified by the option *H5F_LIBVER_LATEST*. In Python, the file has to be opened by `h5py.File(..., libver="latest")`.

The full robot model (see section 3) is used as test case. This model has about 7000 variables, where 2500 are parameters and constants, 800 variables are time varying and the other variables are alias or negated alias variables. For the performance test 500 fixed grid result points and 2·50 varying grid result points due to 50 state events are taken into account. Discrete variables are only stored at event points, but continuous variables are stored at grid and event points, here at 600 points. This gives the sizes of the files in Table 2.

Table 2: File sizes of the RobotR3 example for 500 grid points and different formats. The first column of *Relative Size* is normalized to the result of the Dymola format. The second column is normalized to the results of the MTSF format.

Format	**Raw**	**MB**	**Relative Size**	
Hierarchical variables format with standard options	HDF5	27.6	5.11	7.46
Hierarchical variables format with options *compact* and *latest*	HDF5	7.4	1.37	2.00
Dymola format	MAT	5.4	**1.00**	1.46
MTSF format	HDF5	3.7	0.69	**1.00**

The MTSF format results in a file size that is just half of the file size of the HDF5 hierarchical variables format, so it is clearly superior. Furthermore, the MTSF format gives about 30% smaller file size with respect to the Dymola file, although more meta information is stored than in the Dymola file.

Table 3: File sizes for 5000 grid points and different formats. The first column of *Relative Size* is normalized to the result of the Dymola format. The second column is normalized to the results of the MTSF format.

Format	**Raw**	**MB**	**Relative Size**	
Hierarchical variables format with standard options	HDF5	54.1	1.56	1.79
Hierarchical variables format with options *compact* and *latest*	HDF5	33.8	0.98	1.12
Dymola format	MAT	34.6	**1.00**	1.15
MTSF format	HDF5	30.2	0.87	**1.00**

For result files with increasing number of result points, the relative differences between the different

approaches is decreasing, which can be seen in Table 3 for 5000 fixed grid result points and 2·50 varying grid result points at events. The reason of a decreasing difference are the file size dominating data matrices that are identical at least in the HDF5 files.

4.3 Writing and Reading of Large Files

The previous tests evaluated writing of HDF5 files. If the HDF5 file becomes very large, it can no longer be read in one piece. Reading files which are larger than the main memory is slow, as virtual memory paging has to be used. The question arises how this is handled. In Dymola, and many other simulation programs, reading a result file requires to read it completely in to memory and then the file sizes that can be handled are restricted by the respective main memory. Here the power of the HDF5 format is applied. It is possible to read just a specified column of a matrix, without reading the whole matrix. Internally, the HDF5 matrix is split into chunks (= smaller matrices) and only the relevant chunks are read [THG11].

Performance of writing and reading some parts of a huge matrix depends on amongst others the sizes of the chunks. Because it is not fixed what parts of result matrices are read after writing, the chunking details are not specified for an MTSF file.

Table 4: File sizes and performances of writing and reading MTSF files.

# Rows	# Columns	MB	GB	Writing [s]	Reading 1 [s]	Reading 2 [s]
$6{\cdot}10^3$	766	35.5	0.03	0.5	0.15	0.02
$6{\cdot}10^4$	766	352	0.34	5.9	0.23	0.06
$6{\cdot}10^5$	766	3517	3.4	62.2	0.84	0.2
$6{\cdot}10^6$	766	35160	34	1109	4.9	1.2
$3.6{\cdot}10^7$	766	210410	205	11400	25.5	1.9

In Table 4 experiments with the full robot model on a solid state disk (on a system with an Intel Xeon X5550 @ 2.67 GHz processor) are documented. The number of time points has been increased to get large HDF5 files. Performance of reading two columns (time and one model variable) of the matrix *Results/Continuous/H5T_NATIVE_DOUBLE* into Python is documented in column *Reading 1*. Performance of reading the last row of the matrix (final value of all variables) is shown in column *Reading 2*. We did not investigate how different chunk sizes influence the result. It is clear that a fine tuning can improve the numbers in Table 4.

This test proves to be able to write data to and read it from result files beyond 200 GB in acceptable time. Further tests should verify the handling of huge files. Using HDFView, the structure of large files can be inspected without problems. Only for the 205 GB file, HDFView is slowing down.

4.4 Reading by Matlab

Matlab [M12] is one of the most commonly used scripting tools in engineering applications. Therefore it has to be simple and fast to read data from MTSF files in Matlab. The test concentrates on reading the names of all variables of a result file. Using this list of variables a variable tree browser could be generated. We investigate reading two files of the full robot model. One file is according to the proposed MTSF format (see Section 3), the other file follows the hierarchical variables concept (see Section 4.1).

Table 5: Time for reading all variable names in different formats. For the hierarchical variables concept we distinguish between a format that includes HDF5 enumerations in attributes of HDF5 datasets and replacing them by simple integer values.

Matlab function	Hierarchical Variables Concept		MTSF
	Enum.	Integer	
`h5info`	Error	75 s	0.1 s
`hdf5info` (outdated)	13 s	5.5 s	0.1 s

Matlab 2011b offers the high level functions `h5info` for reading the structure of an HDF5 file and `h5read` for reading one dataset. To get the names of all variables for the hierarchical variables concept one has to read the tree structure of the HDF5 group *ModelVariables*. We use `h5info` for it. Apparently, Matlab is not able to read enumeration attributes in HDF5 datasets. Therefore, we generated a new result file and replaced enumerations by simple integer values. The elapsed time for reading the different files are listed in Table 5. Using the outdated Matlab function

DOI 10.3384/ecp12076495

`hdf5info` we were able to reduce the elapsed time for the hierarchical variables concept.

The MTSF file contains only a few HDF5 groups and datasets, whereas the file of the hierarchical variables concept includes many (small) groups and objects. So it seems evident that reading the result file structure is faster for the MTSF file. To get all variable names from a MTSF file one has to read the dataset *ModelDescription/Variables*. Using the Matlab command `h5read('fullRobot.mtsf', '/ModelDescription/Variables')` the information is available. The execution time for this command is 0.02 s. In summary, reading the variable names from the MTSF file is much faster than for the hierarchical variables concept. These preliminary tests with Matlab also clearly indicate that the proposed file format is better suited than the hierarchical variables concept. Furthermore, the Matlab h5read m-file does not support region references. Besides the other drawbacks discussed in Section 3.1, it is therefore advisable to not use region references in HDF5 files, if the files should be read by Matlab.

5 Conclusions

A standard for time series result files typically generated by dynamic model simulations is proposed. The standard is based on the HDF5 file format because HDF5 offers many features to flexibly and efficiently store data. In test cases huge files larger than 200 GB are successfully written and read. We hope to come into discussion with all persons who are interested in a standard result file format. The goal is to define an internationally well accepted standard that is supported by many tool vendors.

6 Acknowledgements

We acknowledge the coding and testing work of J. M. Solis Lopez' (formerly Bausch-Gall GmbH). M. Friedrich (Simpack AG) gave very useful information to reduce the file size of HDF5 files generated when using the hierarchical variables concept. We acknowledge his support. Also, we are grateful for the constructive comments of the reviewers.

References

[A12] Association for Standardisation of Automation and Measuring Systems. www.asam.net.

[BP11] Bausch-Gall I. and Pfeiffer A.: *Standard efficient Storage of Simulation Results.* ASIM2011, 21. Symposium Simulationstechnik, 7. - 9. Sept. 2011, Winterthur, Switzerland, 2011.

[DS12] Dassault Systèmes AB: *Dymola.* www.dymola.com.

[H12] H5py. http://pypi.python.org/pypi/h5py.

[M12] MathWorks: *Matlab*. www.mathworks.com/products/matlab.

[MA10] Modelica Association: *Modelica Standard Library 3.2*, Oct. 2010. www.modelica.org/libraries/Modelica.

[MC10] MODELISAR consortium: *Functional Mock-up Interface for Model Exchange, Version 1.0*, 2010. www.functional-mockup-interface.org.

[MC12] MODELISAR consortium: *Functional Mock-up Interface for Model Exchange and Co-Simulation*, Version 2.0 Beta 3, 2012. www.functional-mockup-interface.org.

[NCS+12] http://access.ncsa.illinois.edu/Releases/-05Releases/07.12.05_NCSA%27s_HDF.html

[PHH+12] Pfeiffer A., Hellerer M., Hartweg S., Otter M. and Reiner M.: *PySimulator – A Simulation and Analysis Environment in Python with Plugin Infrastructure*. Accepted for publication in the Proceedings of 9th International Modelica Conference, Munich, Germany, Sept. 2012.

[PA11] Phillips A. W. and Allemang R. J.: *Requirements for a Long-term Viable, Archive Data Format*. Structural Dynamics, Conference Proceedings of the Society for Experimental Mechanics Series, Volume 12, pp. 1475-1479, Springer, New York, 2011.

[P10] Poinot, M.: *Five Good Reasons to Use the Hierarchical Data Format*. Computing in Science & Engineering, Vol. 12, Issue 5, pp. 84-90, 2010.

[P12a] Python. www.python.org.

[P12b] Python(x,y). www.pythonxy.com.

[THG11] The HDF Group: *HDF5 User's Guide*, HDF5 Release 1.8.8, Nov. 2011.

[THG12a] The HDF Group, www.hdfgroup.org.

[THG12b] The HDF Group: *HDFView*. www.hdfgroup.org/hdf-java-html/hdfview.

[TSC12] The Scilab Consortium: *Scilab*. www.scilab.org.

DOI
10.3384/ecp12076495