`recon` – Web and network friendly simulation data formats

Michael Tiller
Xogeny Inc., USA
`michael.tiller@xogeny.com`

Peter Harman
CyDesign Ltd., UK
`peter@cydesign.com`

Abstract

There are many different commonly used file formats for storing time series data. Most of these file formats are designed with the assumption that the file itself will be locally available to the software that will be reading or writing the data stored in them. While this assumption is an excellent one for desktop based tools with direct access to disk drives capable of moving virtually instantaneously around from sector to sector, there are a growing number of applications for which local access is not necessarily available. For these applications, we've initiated the `recon` project to develop more suitable formats.

With the emergence of web and cloud based modeling and simulation technologies, the time has come to explore file formats specifically optimized for non-desktop applications. In this paper, we present a new set of file formats that are specifically designed for web and cloud based approaches. This paper reviews the key requirements for web and cloud enabled applications and then presents a specification for two file formats that address those requirements.

When considering the various use cases that drove our requirements, we recognized that two different file formats were actually required. The first format, the `wall` format, is optimized for writing. The other format, the `meld` format, is optimized for reading over a network (*i.e.,* minimizing the number of reads and bytes read). We will discuss the layout of each of these formats and describe the use cases for which they are most appropriate.

In the open tradition of the Modelica Association, the authors have made specifications and implementations for these formats available as open source libraries with the hope that they will benefit the community as a whole.

Keywords: *Modelica, FMI, simulation results, cloud, web, open source*

1 Introduction

Several groups have examined the issue of standardized file formats[1, 2] in the context of Modelica. In keeping with the principles of the Modelica Association, an ideal choice would be a production ready format that is open source and cross-platform. With these requirements in mind, most people consider HDF5 a natural choice. There are already open source implementations and the file format has been widely used. In fact, it has even been adopted by The MathWorks for use in MATLAB.

But HDF5 has some practical drawbacks. The first is that it is not truly cross-platform. The reference implementation of HDF5 is written in C. The implementation is primarily targeted for use within C, C++ or Fortran applications. While there are various libraries available for reading HDF5 on the Java platform[3], they are incomplete and awkward to use.

Another issue with HDF5 is that for simple time-series data it is over engineered. HDF5 is feature rich, of that there is no question, but these features come at the cost of complexity. This is why you see very few implementations outside of the reference implementation. Furthermore, the file format makes extensive use of "seek" operations and assumes they are relatively inexpensive. This assumption is reasonable if you are able to communicate directly with the hard drive that the files are stored on, but it isn't reasonable when these files are only available through the network.

There are, of course, many standards for encoding data in web or cloud based environments. The most popular formats, by far, are Javascript Object Notation (or JSON, for short) and XML. There are other approaches as well like Google Protocol Buffers[4], Avro[5] and Thrift[6]. Different approaches have different goals. Some define schemas to add a level static checking. Others exist mainly to compress the data being transmitted. Finally, others exist to introduce a layer of interoperability with RPC frameworks or "big data" tools like Hadoop[7] and

Storm[8].

So where does that leave us? Should we adopt the tried and true standards from the engineering world and simply live with their lack of interoperability with important platforms like Java or Javascript and poor performance when accessed remotely? Or, should we adapt tools from the "big data" world, that were developed for quite different use cases, to work in the engineering world.

In some sense we've chosen a compromise. As we will see shortly, the `wall` and `meld` formats are fundamentally derived from the `msgpack`[9] specification. This gives us excellent cross platform compatibility. But `msgpack` is simply a serialization protocol. To address some of our more important concerns, to be discussed shortly, we needed to design a format the imposed additional structure on top of `msgpack`. So in this sense, we've created a set of original file formats that leverage open standards, like `msgpack`, but re-purpose them for modeling and simulation applications.

2 Goals

After independently reviewing various file formats, the authors were not happy with the existing options for web and cloud based modeling and simulation. The `recon` project started as a discussion about requirements. For our applications, the following requirements were identified:

2.1 Requirement 1 - Adding Data

Simulations are constantly producing additional data. For this reason, adding new data to an existing file is an operation performed many times during a simulation. For this reason, adding data to an existing file should be fast and easy. The key thing is to avoid having to rewrite previous data or, even worse, move data around within the file. For this reason, the ideal solution is to have the ability to simply append new data at the end of the file.

2.2 Requirement 2 - Minimizing I/O

In web and cloud based applications, it is not always practical to download the complete set of results for a simulation into the browser environment. There are many use cases where it would be best to be able to access results "on demand". In these environments, such requests for data will be done via HTTP[10]. However, each of these requests will come with far greater latency than a simple request to read from a disk drive and far less bandwidth. As such, we would want to minimize the number of such requests and the amount of data necessary to transmit in each request. This means we need a way to "cluster" the data we are interested in so as to minimize both the number of requests required and the amount of data in each response.

2.3 Requirement 3 - Cross Platform Support

The file formats developed as part of the `recon` project are targeted at web and cloud based applications. Client side web programming is dominated by Javascript. On the other hand, server side programming is done in a wide variety of languages (*e.g.,* Java, Python, Javascript). Meanwhile, numerical analyses such as simulation are typically done in languages like C, C++ and FORTRAN. For this reason, it should be possible to implement libraries in all of these languages for generating and extracting simulation results.

2.4 Requirement 4 - Aliasing

One of the common patterns in component-oriented modeling approaches like Modelica is that many variables end up with exactly the same solution trajectories. When storing simulation results from such tools, it is useful to recognize that considerable disk space can be saved by recognizing the fact that these variables all share a common underlying solution trajectory. Typically, the data for each unique solution trajectory is stored once and each variable is simply a reference to the underlying solution trajectory. Even more storage can be saved by recognizing that some trajectories are related to other trajectories by very simple transformations (*e.g.,* a simple sign change). For this reason, it is very useful if these kinds of relationships are directly represented in the file format itself.

2.5 Requirement 5 - Data Types

When dealing with simulation results that come from the solution of differential equations, the main type of result is a solution trajectory. In these cases, both the dependent and independent variables are typically represented as floating point numbers.

But these are not the only types of data a simulation or other numerical analysis might yield. From the Modelica world, we might easily have results that are either reals, integers, booleans or strings (since these are all fundamental built-in types in Modelica). But why not hierarchical data structures (as represented by records in Modelica) as well?

2.6 Requirement 6 - Metadata

One issue with data files is that if you don't provide a means for associating metadata with entities in the file, the metadata will **become** entities in the file. For this reason, we deemed it important that metadata should be treated in a "first-class" way. Specifically, it should be possible to associate metadata with the file as a whole and with data structures in the file all the way down to individual signals. This would allow tools to persist other important information, beyond the solution, in these data files. For example, information about common plots or plotting options, descriptions of the signals, units or display units could all be managed in a structured way without being confused with data and without needing to be a formal part of the file format specification.

2.7 Requirement 7 - Hierarchy

Many tools create structures that are hierarchical. In the Modelica world, we have deep hierarchies of instances in simulated models. We also have hierarchies for packages and the definitions contained in them. So it is important that a file format can represent these hierarchies in some way.

In our experience, trying to organize results according to an instance hierarchy creates quite a bit of complexity. While tools could exploit some of the previous requirements (primarily 5 and 6) to achieve a hierarchical representation, we've found that simply encoding hierarchy in the names of variables (*e.g.,* `car.engine.crankshaft.tau`) is typically sufficient and can avoid considerable complexity.

2.8 Requirement 8 - Easy Translation

Even though our goal is to have a format that is well suited to web and cloud based applications, it should also be capable of representing the kinds of simulation results we interact with in a desktop environment. For this reason, one of our requirements was the ability to translate data in the "dsres" format into the `recon` formats. Such a translation should preserve all data and metadata normally associated with the "dsres" format. Furthermore, the resulting `recon` files should be approximately the same size as the original "dsres" file.

3 Approach

3.1 Reading vs. Writing

In reviewing these requirements, the main design challenge was trying to reconcile requirements 1 and 2. Implementing requirement 1 typically involves the need to write data out one row at a time, where each row represents the values of all the variables for successive solution times. As such, the solution values for any particular *variable* are widely spaced. However, requirement 2 requires us to be able to extract a given variable with a minimum number of I/O operations. In other words, requirement 1 typically results in data being fragmented while requirement 2 depends on that data being clustered together.

Our solution to this design problem was to design two file formats. The first, the `wall` format, is designed for writing. Not only does it make adding data fast and efficient, it also supports, unlike the `dsres` approach, adding data for multiple tables at once. In practice, this means that if you have variables in your simulation that are partitioned such that they have different independent variables (as with the new clock semantics in Modelica 3.3), this format supports writing out new data for any of these variables. In other words, you can add results that may have completely different time bases. You can also include results from multiple simulations and use the metadata features to associate the specific parameter sets with each table.

The other format, the `meld` format, is optimized for reading. Specifically, it is optimized for requirement 2. For an ideal format, it should be possible to extract a single result trajectory in a single read. This would act to minimize the impact of latency. Furthermore, the bytes read should contain only data associated with the desired signal. This would minimize the impact of limited bandwidth. As we will discuss shortly, the `meld` file manages to achieve these performance characteristics for all but the first signal read.

Our expectation is that tools will write data out (during simulation) in the `wall` format. Tools may choose to keep the data in this format. For platforms where network access is not a requirement, the write optimized nature of the `wall` format will probably be adequate for both reading and writing. However for cases where data will be read over a network, we expect that tools will, upon completing a simulation, rewrite their data into the `meld` format.

3.2 Serialization

There are really two aspects to each `recon` format. The first is the structure of the file (where different pieces of information reside in the file, something we'll discuss in Section 4) and the other is how the actual data is represented.

Obviously, the data is represented as individual bytes. So we must define the process by which multi-byte pieces of information (*e.g.,* floating point numbers) are "serialized" into bytes. One of the implicit goals of this project was to make a file format that was easy to read and write. Since serializing and deserializing data was a big part of the implementation, we could make the implementation much easier if we leveraged existing standards for serialization and deserialization.

One of the interesting things about coming from the web and cloud based application side is the ubiquity of JSON notation. While the Javascript language itself has many "unusual" semantics, the syntax and semantics around serialization and deserialization are surprisingly simple and intuitive. Unlike XML, for example, writing a parser for JSON and then mapping into a native language representation is surprisingly easy and widely supported.

However, JSON is a textual representation. The problem with a textual representation is the additional overhead of having to parse and interpret the text and convert, without any loss of precision, into a binary representation. For this reason, we didn't consider JSON by itself a practical approach to serialization and deserialization.

While we wanted to avoid the parsing aspect of JSON, the JSON data model[11] is well suited to our purposes and many different groups have attempted to create a binary representation that follows the JSON data model. So our initial approach was to consider BSON [12] which is a binary format that is formally specified and widely implemented because it is one of the cornerstones of the MongoDB database[13].

Unfortunately, the BSON serialization scheme has a significant drawback. The way it serializes arrays is very space inefficient. This is because JSON itself supports sparse indexing of arrays. As a result, a serialization must include, for each value in the array, the index as well. This adds significant overhead. There is no way to specify that all elements in the array are sequential. As such, there is no way to avoid this significant penalty.

Fortunately, our reference implementation in Python[14] had a clean separation between the serialization scheme and the structural aspects. This made it very easy for us to experiment with other serialization techniques. We investigated other similar serialization schemes like Smile[15], BJSON[16] and UBJSON[17]. However, none of them seemed to have a critical mass behind them. Indeed, there doesn't seem to be a consensus in the JSON community on how to serialize JSON in a binary form.

As part of our investigation, we also looked at `msgpack`. It turned out that, like BSON, `msgpack` was formally specified and implemented for a wide variety of platforms[9]. Furthermore, in testing `msgpack`, we found that it had much better storage efficiency compared to BSON. So, in the end, we moved forward using the `msgpack` serialization scheme.

The `msgpack` approach had a couple of unanticipated benefits. In `msgpack`, floating point numbers can be encoded in either single or double precision representations. Also, the specification identifies and formally specifies several different optimizations to minimize the number of bytes required to store short integers or short strings. The underlying "types" permitted in this format map very easily into the JSON format which, in turn, means that it maps well into the native data types common across all the languages we are interested in. Finally, the `msgpack` serialization scheme includes an extension mechanism for including additional data types beyond those in the specification. While we don't have any immediate use for these extensions, it is nice to have that feature if we ever find that `msgpack`'s serialization is too constraining.

4 Specification

With the motivation behind us, let's turn to the actual specification of these formats. In this section we will describe the layout of both the `wall` format and the `meld` format. As mentioned in the previous section, the serialization is done using `msgpack`. So we will focus mainly on the layout of data within the file. When describing the actual data being stored we will use JSON notation to document the data with the implicit understanding that this data will be serialized and deserialized using `msgpack`.

Before we get into the specifications for each of these formats, it will be useful to discuss a few topics that are relevant to both formats. For example, what exactly are we storing in these file formats? Both formats support the storing of tables and objects. Tables are, as the name implies, a structure for storing tabular data. It is worth noting that there are no restrictions on what kind of data can be stored in a table except those imposed by the underlying `msgpack` format (which are very few). This means tables can mix integers, floats, doubles, longs, short ints, strings, booleans and even objects across different columns in the same table.

In addition to tables, we can register objects to be stored in our results file. These are essentially free-format pieces of data. The objects can be used to create arbitrarily deeply nested data structures that mix all varieties of data. Once again, the only limitations are those imposed by `msgpack` and/or the source language.

When storing tables and objects in a file, they must be referred to by names. The same namespace is used for both objects and tables (in other words, you cannot have a table with the same name as an object). It also means that no two tables and no two objects can share the same name either. The columns of each table are also named and no two columns within the same table can share the same name. But there is no such restriction between columns in different tables (or fields in different objects, for that matter).

Finally, it is worth noting that certain pieces of data are optional. In those cases, we have consistently followed a policy of leaving out both the key and the value. In other words, it is not sufficient to simply associate a `null` value with a key. **Both the key and the value must be removed** if there is no value provided. The guiding principle here is that parsers should not have to do excessive amounts of null value checking.

With that background out of the way, let's proceed with our explanation of the two file formats.

4.1 Wall Format

Recall that the `wall` format is optimized for writing and that this, in turn, means being able to easily add data. You can think of the `wall` format as being similar to a brick wall where each brick (new piece of data) is staggered with respect to others. As you will see, we are **not** storing information in homogenous arrays and this means that we cannot predict the index of data simply based on information about which row or column it is in. Also note that it is possible to have data from two different tables interleaved between each other. This allows us to add data with two distinct time bases. But it makes the location of data even more difficult to predict. However, remember that the `wall` format is optimized for writing, not reading and that if network access is required then tools will typically rewrite their data into the `meld` format.

4.1.1 Leading Bytes

Each `wall` file starts with the following sequence of bytes:

```
0x72 0x65 0x63 0x6f 0x6e 0x3a 0x77
0x61 0x6c 0x6c 0x3a 0x76 0x30 0x31
```

This is a hex encoding of the ASCII string `recon:wall:v01`. This allows us to identify whether this is a `recon wall` file and, if so, what version of the specification should be applied.

The next four bytes are a binary encoding of the length of the header. This encoding is done in so-called "network byte order" (big-endian). The byte encoding of the length is not considered part of the header (*i.e.,* the length indicated doesn't include the 4 bytes that encode the length).

4.1.2 Header

Once the length of the header is known, the bytes for the header are read in. These bytes are assumed to have been serialized in `msgpack` format so we must next unpack (deserialize) these bytes. Once unpacked, the header should contain the following information:

```
{
  "fmeta": {<file-level metadata>},
  "tabs": {
    "<table name>": {<table data>}
  },
  "objs": {
    "<object name>": {<object metadata>}
  }
}
```

The `"fmeta"` key is associated with the value for any file level metadata. This metadata is, itself, represented in our notation here as an object in JSON but will be encoded as a map in `msgpack`. The `"tabs"` key is associated with a value that maps the names of tables to table data. The format of this table data is as follows:

```
{
  "tmeta": {<table-level metadata>},
  "sigs": [<list of signals>],
  "als": {
    "<aliasname>": {
      "s": <base signal name>,
      "t": <transform string> // OPTIONAL
    }
  },
  "vmeta": {
    "<varname>": {
      <variable-level metadata
    } // OPTIONAL
  }
}
```

The `"tmeta"` key is associated with metadata (again, represented as a `msgpack` map) but this time it is metadata associated with the table. In addition, we have the `"sigs"` key which represents an ordered list of signals. A signal represents an actual solution trajectory and the order is important because the order in this list indicates the order in which the data will be stored in successive rows (to be discussed shortly).

The `"als"` key represents any aliases present in the table. Again, this is a `msgpack` map where the name of the alias is the key. There are two essential pieces of information associated with each of the aliases. The first, stored under the `"s"` key (which stands for "signal") is the name of the base signal that this alias is based on. The `"t"` key is used to represent the transformation that should be applied to the base signal to compute the value of the alias signal. Note that this transformation **is optional**. The possible values will be described shortly in 4.3.

Finally we have the `"vmeta"` key, which is a map where the keys are the names of variables (*i.e.,* both signals and aliases) and the values are any metadata (again, stored as a `msgpack` map) associated with the named variable. Note that keys are only present in the `"vmeta"` map if there is metadata associated with that variable.

Returning to the header level entries, the `"objs"` key is associated with a value which is, in turn, a `msgpack` map. Each key in that map represents an object name and the value associated with those object name keys represents the metadata associated with the named object.

4.1.3 Entries

Following the header, the remainder of the file consists of "entries". Each entry is preceded by 4 bytes in network byte order indicating the length of the entry (again, the length indicated does not include the 4 bytes used to represent the length). All entries are encoded maps in `msgpack` format. There are two types of possible entries and there are no rules about which can be present (*i.e.,* they can appear in any order and be interleaved).

The first type is a "row entry" which details a new row for a specified table. The format of a row entry is as follows:

```
{
  "<table name>": [list of signal values]
}
```

where the table name must be a key in the `"tabs"` map found in the header and the order of values in the list of signal values must correspond to the order defined by the list associated with the `"sigs"` key value within that table.

The other entry type is a "field entry". These represent updates to the values of fields in objects and have the following format:

```
{
  "<object name>": {
    "<field name>": <field value>,
    "<field name>": <field value>
  }
}
```

Note that the header does not specify which fields are present in the object. This can only be determined by processing all the field entries and incorporating fields as they are given values. The complete value for a given object is determined by processing all field entries associated with that object in the order they appear in the `wall` file. The complete value is then simply the final value after all processing is completed. One consequence of this is that it is therefore possible to have the values of individual fields change while writing the file.

4.1.4 Wall Format Summary

The following outline attempts to summarize all the details presented so far:

```
// ID
0x72 0x65 0x63 0x6f 0x6e 0x3a 0x77
0x61 0x6c 0x6c 0x3a 0x76 0x30 0x31
```

```
// Header length, network order
0x?? 0x?? 0x?? 0x??
{
  "fmeta": {<file-level metadata>},
  "tabs": {
    <table name>: {
      "tmeta": {<table-level metadata>},
      "sigs": [<list of signals>],
      "als": {
        <aliasname>: {
          "s": <base signal name>,
          "t": <transform string> // OPTIONAL
        }
      },
      "vmeta": {
        <varname>: {
          <variable-level metadata>
        } // OPTIONAL
      }
    },
  },
  "objs": {
    <objname>: {<object metadata>}
  }
}

// Followed by zero or more entries
// which can be either...

// ...field entries...
0x?? 0x?? 0x?? 0x?? // entry length
{
  <object name>: {
    <field name>: <field value>,
    <field name>: <field value>
  },
}

// ...or row entries
0x?? 0x?? 0x?? 0x?? // entry length
{
  <table name>: [list of signal values]
}
```

where all maps are encoded in `msgpack`.

4.2 Meld Format

4.2.1 Leading Bytes

Each `meld` file starts with the following sequence of bytes:

```
0x72 0x65 0x63 0x6f 0x6e 0x3a 0x6d
0x65 0x6c 0x64 0x3a 0x76 0x30 0x31
```

This is a hex encoding of the ASCII string `recon:meld:v01`. This allows us to identify whether this is a `recon meld` file and, if so, what version of the specification should be applied.

The next four bytes are a binary encoding of the length of the header. This encoding is done in so-called “network byte order” (big-endian).

4.2.2 Header

Once the length of the header is known, the bytes for the header are read in. These bytes are assumed to have been serialized in `msgpack` format so we must next unpack these bytes. Once unpacked, the header should contain the following information:

```
{
  "fmeta": {<file-level metadata>},
  "tabs": {
    "<table name>": <table data>
  },
  "objs": {
    "<object name>": <object data>
  },
  "comp": true|false // Compression flag
}
```

This is very similar to the `wall` format presented in Section 4.1. Again, we see file level metadata exactly as it is used in the `wall` format. We also have the `"tabs"` and `"objs"` keys, also present in the `wall` format but with an important distinction which is that the values that follows them have a different format, as we shall see shortly.

But we also have a new key, the `"comp"` key, which isn’t present at all in the `wall` format. The value associated with the `"comp"` key indicates whether the remainder of the file (after the header) is not just encoded (using `msgpack`) but also compressed. If the value associated with the `"comp"` key is `true`, then all remaining `msgpack` encodings present in the file after the header are compressed using bz2[18] compression.

For reasons that will become obvious, the data for tables and objects is different in the `meld` header than in the `wall` header. In a `meld` file, the table data has the following format:

```
{
  "tmeta": {<table level metadata>},
  "vars": <list of variable names>,
  "toff": {
    "<varname>": {
      "i": <index of variable data>,
      "l": <length of variable data>,
      "t": <transform string> // OPTIONAL
    }
  },
  "vmeta": {
    "<varname">: {
      <variable level metadata
    } // OPTIONAL
  }
}
```

The "tmeta" is, again, the table level metadata. Similarly, the "vmeta" key is associated with variable level metadata which is a map where the variable name is the key (again, only present if there is metadata associated with the specified variable) and the associated value is the variable level metadata.

The "vars" key is associated with an ordered list of the variables present in the file. The "toff" key is associated with a map that specifies important information about the location of the variables within the file. It is the "toff" data that makes it easy for us to extract individual signals. The "i" key is associated with the starting byte, within the file (starting from 0), of the data associated with the variable and the "l" key is associated with the length of that data. The optional "t" key defines the transformation, if any, to be applied to the variable data (see Section 4.3 for more details).

Returning to the header data, the object data associated with the "objs" key has the following format in a meld file:

```
{
  "ometa": {<object level metadata>},
  "i": <index of object data>,
  "l": <length of object data>
}
```

The "ometa" key is associated with the metadata of the associated object. The "i" and "l" keys are used just as they are within tables, to define the index and length, respectively, of the object data within the file.

4.2.3 Variable Data

As discussed in Section 4.2.2, both variables (conceptually, columns in tables) and objects have an offset and a length provided in the header. In the case of a variable, the data that is extracted from that location in the file will be a **list** of values in msgpack format. The values in that list represent the values for the specified solution variables (first row first, last row last). In the case of an object, the data that is extracted from that location in the file will be a **map** where the keys in the map represent the fields present in the object and the values associated with those keys are the field values.

4.2.4 Header Size

It is worth pointing out that when writing the file, the exact length of the header (when encoded in msgpack format) cannot be known *a priori* (we shall explain why, shortly). For this reason, a collection of bytes representing the largest possible size must be reserved for the header. The size of the header depends on the number of tables and objects as well as the number of signals in each table so it important that all of this information is known before determining the maximum number of bytes required to represent the header.

However, when the final version of the header is written out (once the location of all the data can be determined), its size may be less than originally anticipated. This is a result of the msgpack format's aggressive compression of small integers. For this reason, there may be a few unused bytes present between the end of the header and the first variable or object data in the file. While it is true that a few bytes will be wasted as a result, it really only means that msgpack's aggresive optimizations will be wasted in this case (and this case only).

4.2.5 Meld Format Summary

The following outline attempts to summarize all the details presented so far:

```
// ID
0x72 0x65 0x63 0x6f 0x6e 0x3a 0x6d
0x65 0x6c 0x64 0x3a 0x76 0x30 0x31
// Header length, network order
0x?? 0x?? 0x?? 0x??
{
  "fmeta": {<file-level metadata>},
  "tabs": {
    "<table name>": {
      "tmeta": {<table level metadata>},
      "vars": <list of variable names>,
      "toff": {
        "<varname>": {
          "i": <index of variable data>,
          "l": <length of variable data>,
          "t": <transform string> // OPTIONAL
        }
      },
      "vmeta": {
        "<varname">: {
          <variable level metadata
        } // OPTIONAL
      }
    }
  },
  "objs": {
    "<object name>": {
      "ometa": {<object level metadata>},
      "i": <index of object data>,
      "l": <length of object data>
    }
  },
  "comp": true|false // Compression flag
}

// Followed by any padding
// resulting from header shrinkage

// Followed by zero or more blocks
// of msgpacked data representing
// either vectors or objects whose
// offsets and lengths are specified
// in the header)
```

4.3 Transformations

In the previous sections, there were several mentions of a so-called "transform string". This is an optional piece of information associated with an alias (in the case of the `wall` format) or a variable (in the case of the `meld` format). It defines the transformation, if any, that must be performed over some base data in order to retrieve the true value of the referenced data. There are presently only two allowed transformation types that are supported by the `recon` formats.

The first transformation type is the "inverse" transformation. This transformation is indicated when the transform string has a value of `"inv"`. The impact of the inverse transformation depends on the data type. For numeric data types, the inverse transformation causes the data to have its sign inverted. For boolean data, the inverse transformation applies a logical not operation to the data. With this simple transform alone, it is possible to avoid storing a significant amount of data.

The other transform type is the "affine" transform. This transformation is indicated when the transform string has a value of `"aff(s,o)"`, where `s` represents a scale factor and `o` represents an offset value. In the presence of this transformation, all **numeric** values in the base data should be multiplied by the scale factor, `s`, and then added to the offset value, `o`. As one reviewer of this paper noted, unit transforms are almost always affine in nature. This means that the affine transformation permits results to be stored in many different physical units without taking up any appreciable additional storage.

No transform should be applied to the data if:

- No transform string is present
- The transform string is unrecognized/cannot be parsed
- The transform does not apply to the underlying data type (*e.g.,* applying the `affine` transformation to a Boolean value)
- There was an kind of error or exception when attempting to apply the transformation

In all but the first case, the tool or environment is strongly encourage to provide an error message alerting the user. Tools are also free to treat all but the first of these conditions as an error and suppress access to the alias data.

5 Discussion

5.1 Use Case

Although it is implicit in the requirements listed in Section 2, it is worth elaborating a bit more on the specific use case that drove these requirements.

For web and cloud based simulation, the bulk of the computational work is done remotely. In cloud services, there is an effect sometimes called "data gravity" which dictates that to improve overall throughput, the computing platforms tends to gravitate to where the data is stored (*i.e.,* the same service provider or at least ones that are connected with low latency, high bandwidth connections).

So if one seeks an optimal configuration where the computing and storage are well connected, it is easy to store the complete simulation results without any significant concerns for latency or bandwidth.

The complication comes from having to access that data via a "normal" network connection. A typical use case (and the one that we imagined while formulating our requirements) is one where a web application, running in a web browser, needs to display simulation results. The question then is how could such an application make effective use of simulations results generated "in the cloud"?

One approach to this could be for the web browser to download the complete simulation results. Without consideration for web and cloud based applications, such results might very likely be stored in formats like HDF5 or MATLAB version 4 format. The first problem is that these formats are not well supported in a Javascript environment. In fact, the authors are not aware of a single Javascript implementation that can read either format.

Even if you had Javascript implementations for these formats, this approach would necessitate having to download the entire file into some kind of "in memory" representation to be parsed. This is because these formats do not provide a simple way to extract the required data via a few simple reads.

The other approach, the one we've taken for the `recon` project, is to use a format (of our own design, the `meld` format) that is designed for remote access. Given access to header information (more on this in a moment), we can extract a single table column or single object with a single read. Furthermore, we can use the HTTP `Range` header to specify exactly which range of bytes we require. In this way, we only have to endure the latency of a single network request and we avoid transmitting any extraneous data which minimizes the impact of bandwidth limitations.

Note the previous paragraph presumes we already have the header information from the files. The worst case scenario for getting header information is to make two additional (one time only) network requests. The first request would be for the first 18 bytes (again, utilizing the `Range` header) to establish the size of the header. The next request would be to read the binary header data which includes information about the locations of all table columns and objects. Once these two requests have been made, the client would be able to access any object or column with only one additional request.

One of the things we've tried to do in this project is avoid solving problems that have already been solved. This was the motivation behind the use of `msgpack`, but we are also assuming that most network requests for data will be made using HTTP. This is a safe assumption given the ubiquitous nature of HTTP (or HTTPS, for that matter). But if we assume that requests are made via HTTP we also gain the additional benefits of caching.

Most networks are already equipped to efficiently handle caching transparently. Of course, within a given application, we might maintain a cache of headers for different results files that we may be interested in. But what about multiple users accessing these results across the same network? As it turns out, once one user accesses the file, it is quite likely that the header information will be stored in a caching proxy between the users network and the storage provider. This is a caching scenario that we cannot address within an application, but nevertheless, it is very likely that in such a multi-user environment the existing network proxies would transparently act to improve overall performance and enhance the user experience.

5.2 Metadata

As discussed throughout Section 4, the `wall` and `meld` formats have extensive support for metadata. This provides a clean mechanisms for including metadata in files without the need to mix it in or conflate it with actual data. Furthermore, metadata is supported for a wide range of entities represented in the file.

There are many potential applications for such metadata. For example, file level metadata can be used to store useful information such as who ran a simulation, what particular model they ran, what modifications (if any) were applied to the model or even a complete listing of all the parameter values that were used to simulate the model. In addition, general experimental data could also be stored in the results file.

Metadata at the signal level could be used to specify not just descriptions of those variables along with their physical units but, could also be used to include other attributes like start values, nominal values, *etc.*,

In summary, the `recon` formats make metadata a first class citizen within the file formats and this opens the door to many very useful applications involving metadata.

5.3 Performance

It is worth taking a moment to discuss the actual performance of this approach vs. existing approaches. As a baseline, we have generated a "representative results file" by simulating the R3 robot example from the Modelica Standard Library. This results file, produced by Dymola and written in the "dsres" format, will serve as our baseline.

The first issue we will examine is space efficiency. The baseline results take up 3,069,623 bytes of storage (for the storage options we selected). When stored in the `meld` format **without compression** those same results take up 3,169,994 bytes. Note that this translation process from the dsres format to the `meld` format preserves variable names and description strings as well as numerical trajectories. This means the `meld` format is only 3.3% larger than the corresponding dsres file. If we enable compression, the `meld` file is only 2,787,467 bytes which means it is almost 10% smaller than the dsres file.

So, in terms of storage, the `meld` format is quite comparable to the dsres format. At first, this may seem counter-intuitive since we know, from our earlier discussion of `msgpack`, that arrays of floating point numbers incur an extra byte for storage. However, the dsres format cannot store strings very efficiently and that long descriptions will result in enormous amounts of padding. So it would appear that, on net, these two effects cancel each other out. However, one advantage that the `meld` format has that the dsres format doesn't (and an advantage that is not capitalized on in these benchmarks at all) is the wider range of alias transformations. In particular, it is possible to relate two signals via an affine transformation with the `meld` format. Unfortunately, we do yet have any data on the potential savings this might offer.

Another benchmark worth considering is the time it takes to extract one particular trajectory from a results file. For extracting results from a dsres file, we used the `scipy.io.loadmat` function to load the `data_2` matrix and then extracted the 0^{th} column (Time). In other words,

```
mat = loadmat("tests/fullRobot.mat")
T2 = mat["data_2"]
time = T2[0,:]
```

Running this script 5 times gave the following times (in milliseconds): 18.111, 18.143, 20.076, 19.309, 21.773 for an average time of 19.482 milliseconds.

In the case of the `meld` file, we opened the results file, created a `MeldReader` object to read it, extracted the `data_2` table (called `T2` in the translated file) and then extracted the data associated with the `Time` signal. That code looks as follows,

```
with open("dsres_robot.mld", "rb") as fp:
  meld = MeldReader(fp)
  dt = meld.read_table("T2")
  time = dt.data("Time")
```

Running this script 5 times gave the following times (again, in milliseconds): 17.260, 18.074, 16.882, 14.608, 14.572 for an average time of 16.280 milliseconds.

It is worth noting the SciPy implementation is very mature and utilizes `numpy` internally. So one would expect excellent performance from that implementation. Nevertheless, extracting data based in the `meld` format was over 15% faster on average.

The last benchmark will be reading multiple signals. In fact, our benchmark in this case will be to extract all transient signals into a Python dictionary. We will do this again for both formats. We used the following script to extract the same signals from the dsres file format:

```
ret = {}
mf = dymat.DyMatFile("tests/fullRobot.mat")
for signal in mf.names(2):
    ret[signal] = mf.data(signal)
ret["Time"] = mf.abscissa(2)
```

This code uses the `dymat` library which, in turn, leverages SciPy and numpy. Running this code 5 times gave the following execution times (in milliseconds): 519.50, 495.38, 483.60, 476.69 and 481.15 for an average execution time of 491.26 milliseconds.

The script for performing this benchmark using the `meld` format looks as follows:

```
ret = {}
with open("dsres_robot.mld", "rb") as fp:
  meld = MeldReader(fp)
  dt = meld.read_table("T2")
  for signal in dt.signals():
      ret[signal] = dt.data(signal)
```

Running this script 5 times, we record execution times (in milliseconds) of: 246.34, 240.66, 225.58, 224.22, 222.65 for an average of 231.89 milliseconds (over 50% faster compared to the dsres version of the benchmark).

In fairness, it is worth pointing out that we generally expect simulation tools to write output results in the `wall` format first and then convert them into the `meld` format. As such, they will incur some penalty as a one-time cost when regenerating their results in the `meld` format and this penalty is not taken into account here. Furthermore, this benchmark doesn't include any writing performance benchmarks (primarily because we have not connected any of these codes to actual simulation tools to measure write performance).

In summary, we do not have any benchmarks that compare write performance. But in terms of storage efficiency, the `meld` file format was only 3% larger when compared to our baseline results in dsres format. On the other hand, in terms of read performance, the `meld` format was between 15 and 50% faster depending on the amount of data to be read.

5.4 Implementations

As already mentioned, there is a reference implementation of both the `wall` and `meld` formats already available in Python[14]. There are also implementations available for both C[19] and Java[20].

In addition, the `wall` format is also now supported by OpenModelica as of `r18784`. The implementation of the `wall` format required 408 lines of code to implement in OpenModelica (this includes implementation of all necessary `msgpack` operations, *i.e.,* no external libraries are used to implement `msgpack` functionality) while the implementation of the "dsres" format uses 656 lines of code.

6 Conclusion

In conclusion, the `recon` file formats support many important features:

1. **easy to implement on a wide range of platforms** - An open source reference implementation is available in Python with implementations in C and Java already planned.

2. **efficient disk and network access** - These formats have been optimized for efficient network performance. But as the benchmarks show, these formats also perform very well for when results are stored locally on hard disks.

3. **first-class metadata** - With metadata available at all structural levels, results files can embed metadata in a clean and practical way. This has the potential to open up many new and interesting applications.

4. **tables containing mixed data types** - All data types are given equal treatment within the `recon` formats. As a result, it is possible to embed a wide variety of data and still maintain all the other benefits associated with these formats.

5. **richer alias transformations** - The `recon` formats support not only the common "inverse" transform, but a richer set of affine transformations. This could lead to further space efficiencies.

6. **efficient treatment of strings** - MATLAB version 4 files are very inefficient for storing collections of strings. By using `msgpack` for serialization, we get very efficient handling of string data.

7. **ability to store objects/structures** - Not all data is tabular. For example, parameter data used as input to simulations is more naturally represented as an object. There are many other examples of potential applications that need to represent data as objects. The `recon` formats even allow these objects to be embedded within tables.

The `recon` formats provide all these benefits without any significant compromise in storage efficiency.

References

[1] A. Pfeiffer, I. Bausch-Gall, and M. Otter. "Proposal for a Standard Time Series File Format in HDF5". In: *Proceedings of the 9th International Modelica Conference* (2012). URL: `http://www.bausch-gall.de/ecp12076495_PfeifferBausch-GallOtter.pdf`.

[2] Jörg Rädler. "DyMat - HDF5 export and other Modelica/Python projects". In: (2013). URL: `http://www.j-raedler.de/2013/01/dymat-hdf5-export-and-comparable-projects/`.

[3] The HDF Group. *HDF-Java 2.10 release*. 2013. URL: `http://www.hdfgroup.org/products/java/hdf-java-html/`.

[4] Google. *Google Protocol Buffers*. 2012. URL: `https://developers.google.com/protocol-buffers/`.

[5] Apache Avro Project Members. *Apache Avro 1.7.6*. 2013. URL: `https://avro.apache.org/`.

[6] Apache Thrift Project Members. *Apache Thrift 0.9.1*. 2013. URL: `https://thrift.apache.org/`.

[7] Apache Hadoop Project Members. *Apache Hadoop 2.2.0*. 2013. URL: `https://hadoop.apache.org/`.

[8] Storm Development Team. *STORM: Distributed and fault-tolerant realtime computation*. 2013. URL: `http://storm-project.net/`.

[9] Sadayuki Furuhash. *MessagePack - It's like JSON but fast and small.* 2013. URL: `http://msgpack.org/`.

[10] R. Fielding et al. *Hypertext Transfer Protocol - HTTP/1.1*. 1999. URL: `http://www.w3.org/Protocols/rfc2616/rfc2616.html`.

[11] JSON. *JavaScript Object Notation*. 2013. URL: `http://www.json.org/`.

[12] MongoDB Development Team. *BSON Specification*. 2013. URL: `http://bsonspec.org`.

[13] Inc. MongoDB. *MongoDB Manual, Version 2.4*. 2014. URL: `http://docs.mongodb.org/manual/`.

[14] Michael Tiller. *recon – Web and network friendly simulation data formats, Python Interface*. 2013. URL: `https://github.com/xogeny/recon`.

[15] Jackson JSON processor project team. *Smile Format Specification*. 2013. URL: `http://wiki.fasterxml.com/SmileFormatSpec`.

[16] Pietrzak Roman. *Binary-JSON (BJSON) format specification*. 2013. URL: `http://bjson.org/`.

[17] Riyad Kalla. *Universal Binary JSON Specification*. 2013. URL: `http://ubjson.org`.

[18] Julian Seward. *Bzip2 Homepage*. 2010. URL: `http://www.bzip.org/`.

[19] Peter Harman. *recon – Web and network friendly simulation data formats, C Interface*. 2014. URL: `https://github.com/harmanpa/crecon`.

[20] Peter Harman. *jrecon – Web and network friendly simulation data formats, Java Interface*. 2014. URL: `https://github.com/harmanpa/jrecon`.