Object-oriented virtual sample library: a container of multi-dimensional data for acquisition, visualization and sharing

Shin-ichi Todoroki†

Advanced Materials Laboratory, National Institute for Materials Science,
Namiki 1-1, Tsukuba, Ibaraki 305-0044, JAPAN

Abstract. Combinatorial methods bring about enormous data not only in size but also in dimension. To handle multi-dimensional data easily, a concept of virtual container for combinatorially acquired data is demonstrated which is called "virtual sample library" (VSL). VSL stores the data hierarchically in the order of (1) coordinates in the sample library, (2) names of the measurements performed, and (3) data obtained from each measurement. Thus, the stored data are accessed intuitively just by tracing this tree-like structure and are provided for visualization and sharing with others. This framework is constructed by the aid of an object-oriented scripting language which is good at abstracting complicated data structure. In this paper, after summarizing the problems of handling data acquired from combinatorially integrated samples and availabilities of software tools to solve them, the concept of VSL is proposed and its structure and functions are demonstrated on the basis of one specific experimental data. Its extensibility as a platform for numerical simulation is also discussed.

PACS numbers: 07.05.Kf, 07.05.Rm

Received 12 April 2004, in final form 26 July 2004
Published 16 December 2004
Meas. Sci. Technol. , 16, 1, pp. 285-291 (2005). [© 2005 IOP Publishing Ltd]
http://dx.doi.org/10.1088/0957-0233/16/1/037
http://www.geocities.com/Tokyo/1406/node2.html#Todoroki05MST285

1. Introduction

We can enjoy the benefit of combinatorial technology only after we can afford to analyze the large amount of data obtained. Although large scale data processing is an everyday technique in other fields such as genomics and geophysical fluid dynamics, there still remain some problems unique to our fields. It appears when we perform several kinds of measurements per one combinatorially integrated sample and examine the correlation among these measurements.

In general, the data of each measurement are stored independently in various formats, thus it is very laborious work to pick up the correlated data measured at a specific position in

† Correspondence author (TODOROKI.Shin-ichi at nims.go.jp)

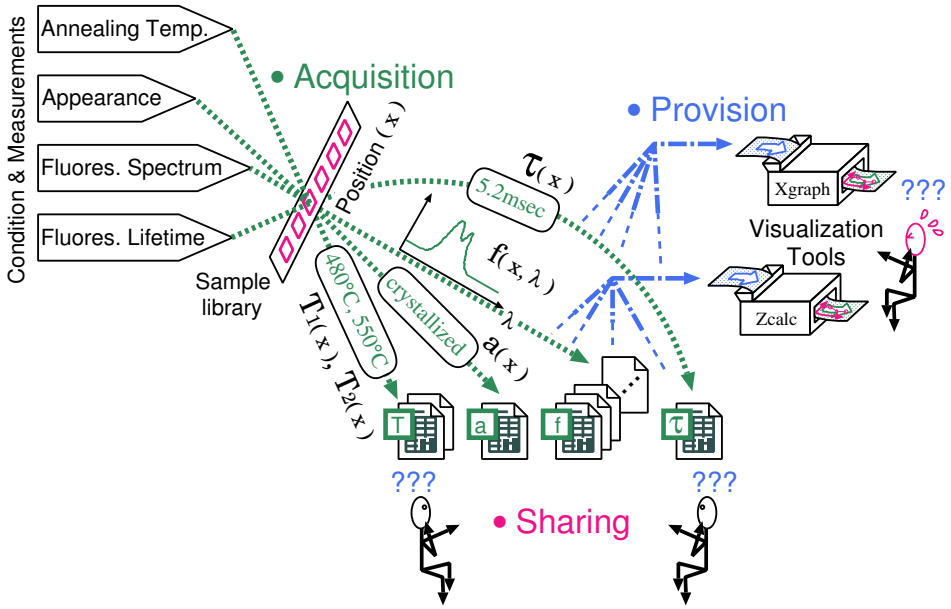

Figure 1. Illustration showing an example of the situation when multi-dimensional data management is needed. Four kinds of measurements over a combinatorially integrated sample[1] bring about an enormous number of data in various formats, which forms 7-dimensional data and are hard to be analyzed and be shared with colleagues.

the sample library (see Fig. 1). Moreover, a number of files in various formats are difficult to be shared with colleagues. In order to solve these problems, all the data should be stored at one place in a common format, i.e. they should be treated as a set of multi-dimensional data. Thus, we need a special software to structuralize and standardize them.

It would be easiest if we can find such software as ready-made or custom-made product, but we often feel not to be willing to pay for them because they do not satisfy our requirements such as compatibility with existing data and/or environments, and cost performance. Therefore, sometimes it is simpler to make software by ourselves, by the aid of modern user-friendly programming languages.

In this context, the author has proposed a concept of 'virtual sample library' (VSL)[2, 3], a container of multi-dimensional data, used for acquisition, visualization and sharing. The most important feature of VSL is that it is written by an object-oriented scripting language, which enables us to treat our complicated data in a simple way. In the former articles[2, 3], however, the author could not give sufficient discussion about why this language was chosen and how the functions of VSL are realized by the language, due to the limited space.

Thus, in this paper, VSLs are fully described including the problems to be solved (Section 2), the reason why object-oriented scripting language is used (Section 3) and their functions (Section 4). Lastly, the extensibility of VSLs is discussed in comparison to existing virtual libraries (Section 5). The aim of this paper is to demonstrate the merits of the present data managing method without any preliminary knowledge of object-oriented programming. For the sake of helping the understanding of its mechanism, its explanation is made on the basis of a specific experimental data[1]. The readers, who want to know the concept of VSL

briefly and quickly, are recommended to read the former and shorter articles[2, 3] first.

2. The root of our problem

The problems we are facing with are summarized in the following three situations.

2.1. Acquisition

In a combinatorially integrated sample, each pixel is identified as the condition it was fabricated, such as composition, temperature, etc. In addition to these, we perform several measurements on it. Since these operations are done separately for the most part, acquired data are recorded independently in various formats, as illustrated in the upper left part of Fig. 1. This situation becomes more complex when some measurements are performed pixel by pixel sequentially and the results are saved separately. Thus, we have to organize many files more than the number of pixels in the sample library.

2.2. Provision

It is quite easy to draw a graph from the data stored in a single file. All you have to do is to specify which data in the file are plotted toward the software you are using. In contrast, when the data for plotting are distributed over a number of files, we have to provide them after re-compiling the files. Such an editing job requires our patience to remember every format of the files needed, as illustrated in the upper right part of Fig. 1.

2.3. Sharing

Dispersed data also causes another difficulty in sharing them with collaborators. People who have not be involved in preparation and/or measurements of the sample find it hard to recognize which files are to be accessed for their needs.

3. Selection of software tools

The solution of these problems is so simple. All the data are to be structuralized and standardized so as to be treated as a single object, as illustrated in Fig. 2. It is, however, easier said than done and there is more than one way to do it. The first thing to do is to select software tools to realize it. In this section, several alternatives are surveyed briefly with their pros and cons, and finally the author's choice is presented. Such a discussion is useful for those who wish to find better way to meet with their own requirements.

3.1. Creating a database or other solution

What we have to note here is that the problems listed above are not resolved simply by creating a database. As far as the data are registered individually into the database, we have to re-

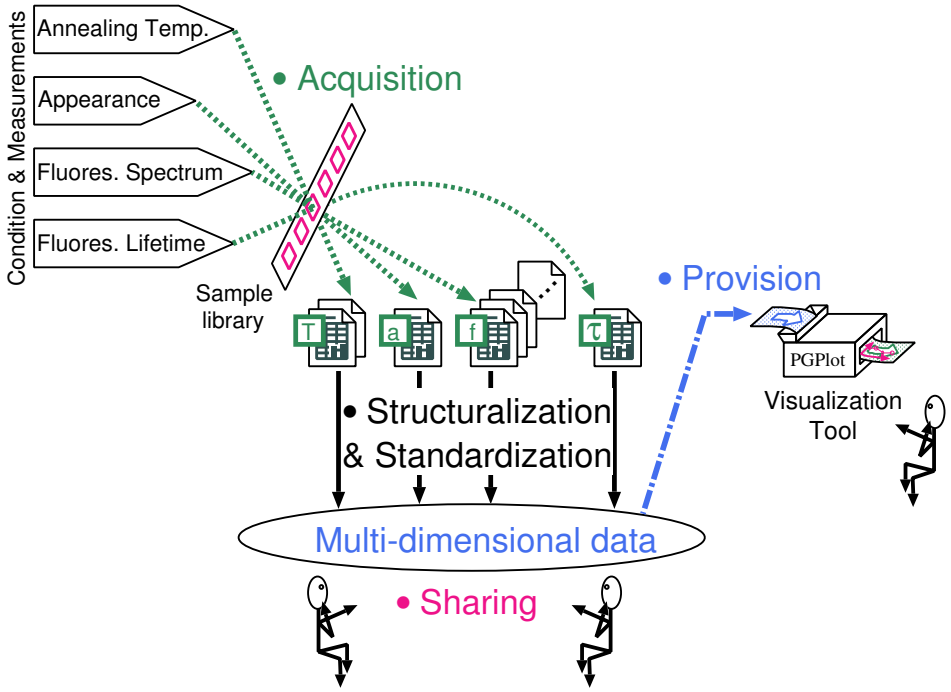

Figure 2. Illustration showing a solution of data flooding. Combinatorially acquired data, which are recorded independently in various formats, are structuralized and standardized so as to be treated as a single object, which is easy to handle and share.

organize them as a multi-dimensional data. Thus, we still need a software for data collection from the database to make a single object of multi-dimensional data.

3.2. Do-it-yourself or outsourcing

There is a bit of truth to say that we as researchers in materials science should outsource such a development of data managing system and devote ourselves to our specialties. It's a trade-off between what we must/can do for outsourcing and do-it-ourselves. For outsourcing, we have to get enough funds, write specifications for system vendors, and wait for the completion of the development. For do-it-ourselves, the size of to-do list depends on our skill for programming, available software tools, and the application area we want to cover. The first two items are complementary; programming skill is less needed when a better human-centered user interface is provided with the software tools. The last item includes data acquisition from experimental apparatus and/or database system, data provision for visualization tools, and data sharing among others.

3.3. Programming languages

History of programming language is the one getting rid of the tasks of programmers to adjust to the convenience of computers and focusing their efforts on more substantial works. Just as assembly languages were expelled from the main stream in the past by high-level

languages such as FORTRAN, Pascal and C, some new languages have been on the rise recently including scripting languages and object-oriented languages[4].

Scripting language is often called as glue language. According to FOLDOC (Free On-Line Dictionary Of Computing, http://foldoc.doc.ic.ac.uk/foldoc/), its definition is, *Any language, ..., used to* write glue *to integrate tools and other programs to solve some problem*. This metaphoric name of "glue" comes from its easy of use like a stationery and its ability to process data into any required format so as to collaborate with existing software tools. In other words, the programs written by this language, called script, are executed without compiling, unlike the conventional high-level languages. Moreover, the language is full of features to make scripts easy to write and read. Perl, one of the most popular scripting languages, is commonly used in bioinformatics[5].

Object-oriented languages provide us a different way to reduce the load for programming. The language has an ability to describe a group of data as an "object" with a set of procedures of how to treat these data. This feature is suitable for us to construct a single object containing multi-dimensional data. Once multi-dimensional data object is generated, we don't have to worry about remembering its inner data structure in detail and can process it just by calling the procedures stored in it. Entry-level users should be, however, ready to spend some time to learn object-oriented programming.

There are some languages having both the two features, called object-oriented scripting language. They include Python, Ruby and Perl, which are open source software (see the later text) and used in other scientific field such as geophysics[6] and bioinformatics[7].

3.4. Extensibility and open source software

Whatever ideal options we discuss, final choice eventually depends on individual circumstances, such as necessity for special functions which are available only by specific software/language. For example, we may need data acquisition via GP-IB / RS232C / parallel / TCP-IP / USB port, data retrieval from databases via Structured Query Language (SQL), or data provision for specific visualization tools. Such features are not necessarily available in every software as it is. We should notice, however, whether they can be implemented by existing external libraries or not. Some software provide such an extensibility by linking a small user-written program to bridge between the software and the libraries.

The ultimate extensibility is realized when the source code of software is open and allowed for the users to improve. "Open source" software is one of the software having this nature. The definition of "open source" is strictly given here[8] in order to promote the development of open source software by utilizing this nature. We are free to improve it, write add-in packages and release them to the public. Thus, we may also find a specific function already implemented by others. Many software packages like GNU/Linux, Apache, Perl etc. have grown up in this framework and become popular because of their high quality and reliability. Other pros and cons in laboratory use are summarized in [9].

Table 1. List of the 7-dimensional data used in this study (see text). Some of these are plotted in Fig. 3 and 4.

#	Data	Fig. 3	Fig. 4
1	Position, x/mm	✓	✓
2	1st annealing temp., $T_1/^\circ$C	✓	✓
3	2nd annealing temp., $T_2/^\circ$C		✓
4	Appearance of glass segment *(White, Opaque, or Transparent)*	✓	✓
5	Fluorescence lifetime, τ/msec	✓	✓
	Fluorescence spectra (CW component)		
6	Intensity (a.u.)	✓	
7	Wavelength, λ/nm	✓	

3.5. The author's choice

As the author's solution, virtual containers of multi-dimensional data are constructed, named virtual sample libraries (VSLs) written by open source object-oriented scripting language, Ruby[10, 11]. Ruby has been developed by a Japanese software engineer since 1993 and was designed to make the users "concentrate on the creative side of programming, with less stress"[11]. In order to visualize the data in VSLs, an external graphic library is used, called Ruby/PGPLOT[12]. It was originally developed for use with astronomical data reduction programs in Caltech since 1983 written in FORTRAN[13]. Thanks to the variety of graphic primitives available via function call, we can design graphs with a high degree of flexibility.

4. Virtual sample libraries

In this section, the structure and functions of VSLs are explained with an example of multi-dimensional data acquired from a series of fluorescence lifetime measurements over three Er-doped glass sample libraries[1], whose segments are arrayed in a row and were annealed in a temperature gradient furnace so as to be heat-treated in three different conditions; (0) no annealing, (1) being annealed for 5min, and (2) another successive annealing at a different position in the furnace for 5min. The sum of the dimension becomes seven, which are listed in Table 1, and the total size of data file is about 8MB in text format. The data obtained from three sample libraries are compiled and plotted in Fig. 3 and 4 using VSLs[2, 3].

Hereafter, some simple Ruby codes are given with plain explanations for those who have no background knowledge of Ruby and object-oriented programming, rather than to be terminologically strict. Skilled readers are advised not to rely on the explanations too literally and to consult the codes directly.

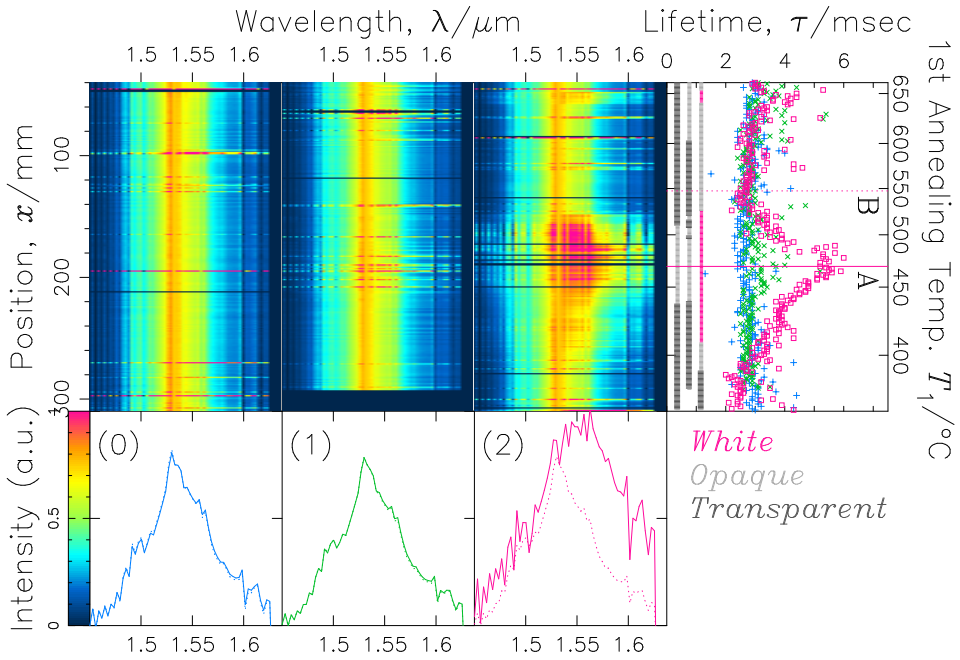

Figure 3. Fluorescence spectra of Er^{3+} (three columns from left, #6 and 7 in the Table 1), lifetime of 1.533nm fluorescence (right, #5) and appearance of glass segment (between the formers, plotted in red, light-gray and dark-gray, #4) plotted along the sample library or as a function of the 1st annealing temperature, T_1. Excitation wavelength is 977nm. (0) with no annealing treatment for reference. (1) with 1st annealing for 5min only. (2) with two successive heat treatments, each for 5min. For the appearance plot, red is for completely crystallized segment, dark gray for transparent, and light gray for opaque. Solid and dashed lines in the lower row and in the right are for the positions named A and B, respectively. The horizontal black solid lines in the intensity plot correspond to the data withdrawn due to their low signal/noise ratio. A lifetime increase is found at around the glass segment A in the sample (2). At the same time, the fluorescence peak at 1.55μm is broadened.

4.1. Structure

VSL models after actual combinatorially integrated sample library. Namely, its logical array structure is similar to the cell structure of the real library. In each cell in the VSL, references to experimental data (or another references to data) are stored. In other words, VSL stores the data hierarchically in the order of (1) coordinates in the sample library, (2) names of the measurements performed, and (3) data obtained from each measurement. Thus, VSL has a tree structure and each experimental data is intuitively accessed by tracing references from the root of the tree. Namely, VSL stores not only experimental data but also the relations among the data and the positions in the actual sample library.

This hierarchy structure is visualized as “pull-down menu” style in Fig. 5. In the bottom of the window, the root menu is located showing the coordinates in the library, from 40 to 60. After the item of “41” is selected, a sub-menu appears showing what kinds of data are stored. Next, the item of “Fluorescence Spectrum” is chosen and another sub-menu is opened to show it consists of two items. Finally, by choosing “Wavelength” item, a series of wavelength values are shown at “Value” box just above the main menu. In this way, such a hierarchy

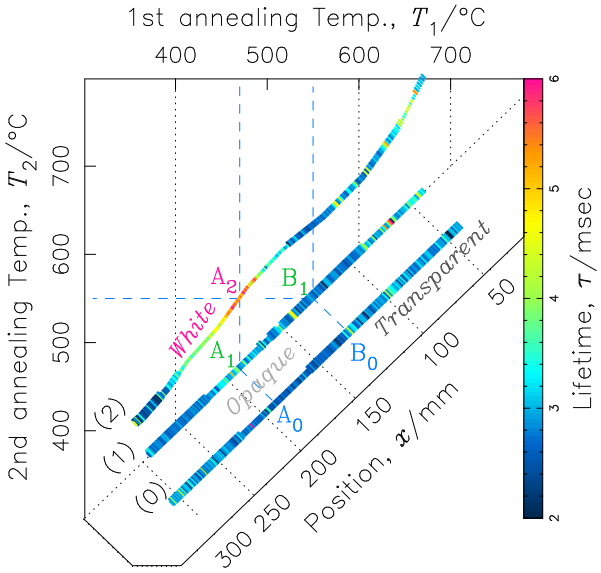

Figure 4. Annealing conditional and/or positional dependence of the lifetime and appearance of the sample libraries. A_n and B_n indicate specific glass segments. These notations are also used in Fig. 3. The maximum lifetime was obtained by annealing at 550°C and then 470°C (A_2).

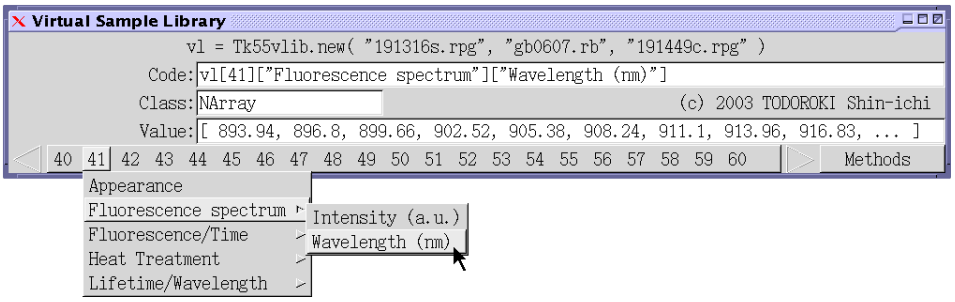

Figure 5. An illustration showing an example of VSL's hierarchy structure by "pull-down menu" style (see text).

structure can store the whole multi-dimensional data systematically and gives us an easy way to access the individual data.

4.2. Provision

Figures 3 and 4 are generated by user-written Ruby scripts, in which three VSL objects are constructed and some functions of Ruby/PGPLOT are called with some extracted data from the VSL objects. The action of tracing data tree shown in Fig. 5 is equivalent to the Ruby code displayed in the "Code" box, where the variable **`vl`** corresponds to one of the VSL objects and the following square brackets mean an operator returning the element referred as the key inside the brackets. Thus, **`vl`**`[41]` corresponds to the list of measurements performed at the pixel of 41 and **`vl`**`[41]['Fluorescence spectrum']` the axes list of fluorescence spectrum at 41.

Followings are the examples of Ruby code used for plotting the extracted data used in Fig. 3. The fluorescence spectra plotted at the bottom of Fig. 3 are generated by the following code,

```
pgline vl[x]['Fluorescence spectrum']['Wavelength (nm)'],
       vl[x]['Fluorescence spectrum']['Intensity (a.u.)']
```

where `pgline` is a function for drawing a line specified with the accompanying two arguments representing x- and y-axes.

Three color images at the top-left of Fig. 3 representing the positional dependence of fluorescence spectra are generated by the code,

```
a = []
vl.each do |v|
  a << v['Fluorescence spectrum']['Intensity (a.u.)']
end
pgimag a, RANGE_OF_Z-AXIS
```

where a two-dimensional array, `a`, is formed during the first four lines and `pgimag` is a function for drawing a color image of the array. The first line defines an empty array named as `a`. The lines from the second to the fourth form a loop executed for each of the elements in `vl`. The variable `v` in the third line corresponds to one of the elements, `vl[x]`. In the loop, an array representing fluorescence spectrum is extracted from VSL and added to `a`, where this operation is described as "`<<`", to form the two-dimensional data array.

Frequently we need another cross-sectional view along an axis other than wavelength, such as the positional dependence of fluorescence lifetime plotted at the top-right of Fig. 3. In this case, a simple procedure picking up the related data are required as listed below.

```
x, y = [], []
vl.each_with_index do |v,i|
  x << v['Lifetime/Wavelength']['Lifetime (ms)'][z]
  y << i
end
pgpt x, y, SYMBOL
```

where `pgpt` is a function for drawing a series of points specified with the accompanying two arguments representing x- and y-axes. The first line defines empty arrays named as `x` and `y`. In the loop from the second line to the fifth, the `z`-th lifetime value, that is, the value at the wavelength of `v['Lifetime/Wavelength']['Wavelength (nm)'][z]` nm, is added to `x` and the pixel number of `i` is added to `y`.

Sometimes we have to plot data in the way not provided by the standard functions of visualization tool, such as Fig. 4 and the positional dependence of appearance plotted between the color images and the lifetime curves in Fig. 3. We can clear it up by the aid of programming language. The sample code for the latter case (Fig. 3) is the following,

```
vl.each_with_index do |v,i|
  case v['Appearance']
  when 'transparent'; pgsci DARKGRAY
  when 'opaque';      pgsci LIGHTGRAY
  when 'white';       pgsci RED
  end
  pgpt1 0.4, i, SYMBOL
end
```

where `pgsci` is a command to set the color for plotting and `pgpt1` a function for drawing a symbol specified with the accompanying two arguments representing x- and y-values. In the loop, the color is set according to the value of `v['Appearance']` and a symbol is plotted at (0.4, `i`). For plotting the data in Fig. 4, the position, the size and the color of each data point is changed according to the properties of #1–#5 in Table 1.

In summary, every data in VSL objects is accessed via a series of brackets operators so as to trace its tree-like structure shown in Fig. 5. This excellent traceability helps us to write concise scripts customized with advanced features.

4.3. Acquisition

The data extraction described above becomes possible only after the VSL object is generated. The code for its generation is shown on the top line of the window shown in Fig. 5, in which three data files are specified. `Tk55vlib` is the name of *class* and **`vl`** is an *instance* of this class. The relation between the two is illustrated in Fig. 6 using a metaphor of a mold and its casting. In the definition of `Tk55vlib` class, it is described there how to read the data files specified with the arguments of `new` operator (or "method", a term used in object-oriented programing) and construct a tree-like structure on the basis of fabrication conditions recorded in these files, i.e. #1–#4 in Table 1. The `new` method returns a VSL object, in other words, an instance of `Tk55vlib` class, corresponding to the data stored in the specified files.

After a measurement on the sample library, we can register the result into the VSL object via user-defined methods. In the present case, `register_fluorescence` and `register_lifetime` methods are defined together with the definition of `Tk55vlib` class. It is described there how to find and read data files and add branches corresponding to #5–#7 in Table 1 to the main tree-like structure. These methods are executed by the following code,

```
vl.register_fluorescence
vl.register_lifetime
```

after which the tree structure shown in Fig. 5 is fully constructed. We can add new methods for data acquisition from other data sources, such as database and measurement equipment, as far as the programming language supports to communicate with these sources.

One thing to be noticed here is that `Tk55vlib` class is only valid for the libraries with one-dimensional pixel array. Thus, whenever a new sample library with different pixel

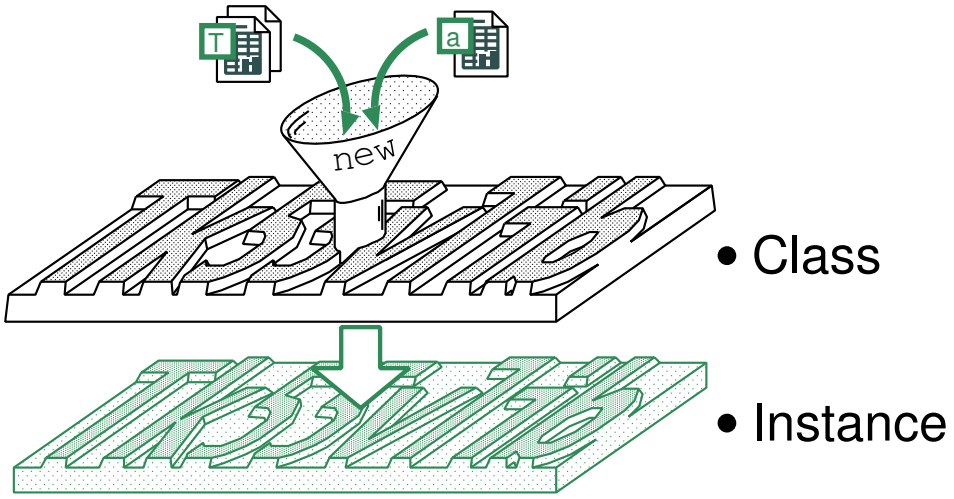

Figure 6. An illustration showing the relation between *class* and *instance* using a metaphor of a mold and its casting. (see text).

geometry is prepared, we have to design a new VSL class for it. It takes, however, not much time for programming since object-oriented languages have a special feature for reusing existing codes with least modifications. Moreover, we can design a new class forming a new tree-like structure consisting of existing VSL objects, which corresponds to an increment of dimensional number. This means that it is possible to treat multi-dimensional data, acquired from a group of "real" sample libraries, as one "virtual" sample library.

4.4. Accessing VSLs over network

Construction of VSL objects takes longer time with increasing the size of multi-dimensional data. Thus, it is not efficient to generate VSLs every time VSL-based script is executed. This problem is solved by using a Ruby library which enables to provide objects over network, called "drb" (distributed Ruby[11]). Under this scheme, one computer acts as a VSL server responding to the requests from client computers, which do not have to spend any CPU time for VSL generation and store any data files. This framework is also used to share VSL objects with colleagues on a secure network.

4.5. Alternative implementation

Since the basic part of VSLs is realized by object-oriented programming technology, VSLs are implementable by other object-oriented scripting languages, such as Python and Perl, conventional object-oriented languages, such as Java, C++, or possibly object-oriented database systems. The choice depends on user's need for the interfaces for acquisition, provision and sharing, i.e. functions for communicating with measurement apparatus, databases, visualization tools and networks.

5. Extensibility beyond data containers

VSLs have a potential to go beyond managing existing data. Since VSLs are objects (please recall the description at the third paragraph of section 3.3), we can add to them some procedures to generate new data or perform simulations by referring the data stored inside. In other words, object-oriented VSLs have functions to perform virtual experiments just like in-silico combinatorial library and virtual combinatorial library[14] do. All these virtual libraries have common characteristic features of storing data for a group of samples and also storing procedures to treat them. The procedures in the present VLSs are focused on acquiring, visualizing, and sharing the data, and the others on simulations. In contrast to the latter, the present VSLs are distinguishable by calling them "object-oriented virtual sample libraries"[3].

6. Conclusions

A concept of virtual sample library (VSL) is presented. VSLs are used for multi-dimensional data management for combinatorial experiments, in which several measurements are performed per one sample library. VSLs enable us to treat the whole data as one object and to access any of the data inside by tracing their hierarchy structure. VSLs are written in open source object-oriented scripting language Ruby, because of its high ability in abstraction and extensibility. Their structure and functions are demonstrated using a specific experimental data.

Acknowledgments

The author express his sincere thanks to Yukihiro Matsumoto (Ruby), Masahiro Tanaka (NArray & Ruby/PGPLOT) and Masatoshi Seki (drb) for providing their excellent software as open source.

References

[1] S. Todoroki and S. Inoue, "Combinatorial fluorescence lifetime measuring system for developing Er-doped transparent glass ceramics," *Appl. Surface Sci.*, vol. 223, no. 1-3, pp. 39–43, 2004.

[2] S. Todoroki and S. Inoue, "Multi-dimensional data management by virtual sample library written in object-oriented script language Ruby," *Transactions of the Materials Research Society of Japan*, vol. 29, no. 1, pp. 293–296, 2004. (Proceedings of The 8th IUMRS International Conference on Advanced Materials, Oct. 2003, Yokohama, Japan, A3-13-O09).

[3] S. Todoroki and S. Inoue, "Management of combinatorially acquired multi-dimensional data through object-oriented virtual sample library," in *Combinatorial and Artificial Intelligence Methods in Material Science II* (R. A. Potyrailo, Q. Wang, T. Chikyow, and A. Karim, eds.), vol. 804 of *Material Research Society Symposium Proceedings*, (Pennsylvania, USA), pp. 327–332, Material Research Society, 2004. (JJ6.7).

[4] J. K. Ousterhout, "Scripting: Higher level programming for the 21st century," *IEEE Computer*, vol. 31, no. 3, pp. 23–30, 1998. (available online at http://home.pacbell.net/ouster/scripting.html).

[5] C. Gibas, P. Jambeck, and J. M. Fenton, *Developing Bioinformatics Computer Skills*. Oreilly & Associates, 2001. (ISBN 1565926641).

[6] “Dennou ruby project.” (http://www.gfd-dennou.org/arch/ruby/).
[7] “Bioruby project.” (http://bioruby.org/).
[8] “Open Source Initiative.” (http://www.opensource.org/).
[9] C. Laird, “Open source in the lab,” *IBM developerWorks*^{TM}, Oct. 2002. (http://www-106.ibm.com/developerworks/linux/library/l-oslab/).
[10] “Ruby Home Page.” (http://www.ruby-lang.org).
[11] D. Thomas and A. Hunt, *Programing Ruby, The Pragmatic Programmer’s Guide*. Addison-Wesley, 2001. (available online at http://www.rubycentral.com/book/).
[12] M. Tanaka, “Ruby/PGPLOT.” http://www.ir.isas.ac.jp/~masa/ruby/pgplot/index.html.
[13] T. J. Pearson, “PGPLOT.” http://www.astro.caltech.edu/~tjp/pgplot/.
[14] C. Suh, A. Rajagopalan, X. Li, and K. Rajan, “Combinatorial materials design through database science,” in *Combinatorial and Artificial Intelligence Methods in Material Science II* (R. A. Potyrailo, Q. Wang, T. Chikyow, and A. Karim, eds.), vol. 804 of *Material Research Society Symposium Proceedings*, (Pennsylvania, USA), pp. 333–341, Material Research Society, 2004. (JJ9.23).