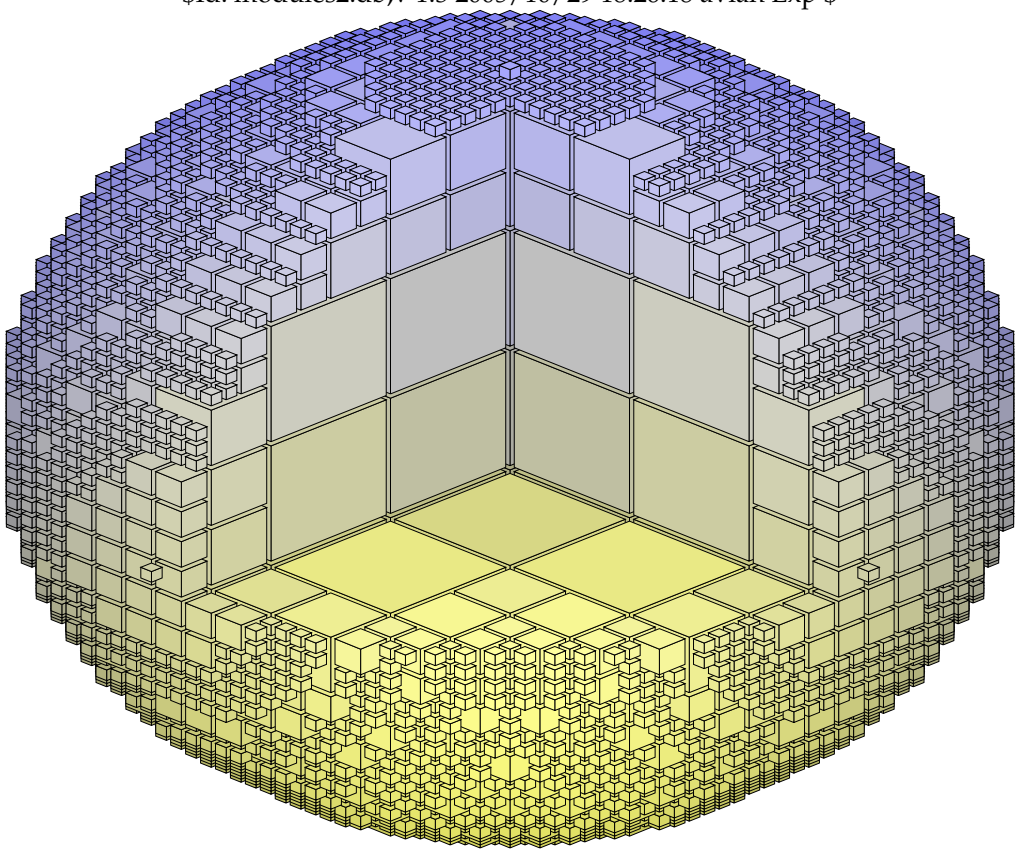
Tablix modules HOW-TO, part 2

Tomaž Šolc

$Id: modules2.db,v 1.3 2005/10/29 18:26:18 avian Exp $



Tablix modules HOW-TO, part 2

Export modules are pieces of code that are dynamically linked with the `tablix2_output` utility at run time and provide functions for exporting data in internal kernel structures to various file formats. This document describes in detail how to write and build new export modules. It also briefly explains how to use kernel API interfaces for this kind of modules.

Table of Contents

Chapter 1. Introduction

Since Tablix version 0.0.6 each file format supported by the `tablix_output` utility is handled by a separate export module. The export module interface changed considerably with the kernel rewrite during 0.2.x branch.

There are two distinct types of modules: Fitness modules contain partial fitness functions and provide handlers for various restrictions. They are loaded by the kernel as specified in the XML configuration file. Export modules on the other hand are loaded by the `tablix2_output` utility and contain functions that translate data from the internal kernel structures to a file in a certain format. For example: HTML export module, comma separated value format export module.

Fitness and export modules access the kernel data structures in more or less the same way. Description of this common interface can be found in the first part of this HOW-TO and will not be repeated here. This document describes only the parts of the interface that are specific to the export modules.

I recommend reading the Tablix User's Manual and the first part of this HOW-TO before attempting to write your own module. Also while reading this text you should keep a browser window nearby with the Tablix kernel API reference manual loaded. Some important functions are also described in the text, but mostly only references to the reference manual will be given.

Chapter 2. Kernel API

2.1. Export function

The export module interface is very simple compared to fitness modules. Each export module must contain only one function, called `export_function()`. Its single purpose is to convert the data stored in the kernel data structures into a stream of characters and store it into one or more files.

Each time user runs `tablix2_output` with the proper command line arguments, the utility loads the requested export module, parses the XML file and initializes kernel data structures and calls the `export_function()`.

The prototype for `export_function()` can be found in `output.h`:

```
typedef int (*export_f)(table *tab, moduleoption *opt, char *filename);
```

tab is a pointer to the table structure. This structure describes the timetable that should be exported - it contains pointers to the chromosome structures.

opt is a pointer to the linked list of module options. Module options are passed to the `tablix2_output` utility with the *-s* argument in the following form:

```
-s option1=value,option2=value,...
```

You can access this linked list in the same way as in fitness modules using functions `option_int()`, `option_str()` and `option_find()`.

The final argument *filename* is a string holding the name of the file to be written. The `tablix2_output` utility simply passes this file name from its command line to the export function. The utility itself does not care if the export function actually writes anything to this file. For example, if the export function needs to write more than one file, it can use this argument as the name of the directory to put the files in. *filename* could also contain the location in a database where the timetable data should be inserted.

> **Note:** *filename* can be equal to *NULL*. The export function should treat that condition as a request to write on the standard output. If this is not possible (i.e. more than one file needs to be written) then the export function should return an error.

The `export_function()` should return 0 on success and -1 on error. Functions `error()`, `info()` and `debug()` can be used to report various warnings and errors back to the user.

2.2. Compiling your module

As with fitness modules there are also two ways of compiling your export modules. You can use your own `Makefile` or you can modify the `Makefile.am` supplied in the distribution and compile your module in the exactly the same way as the official modules. See the first part of this HOW-TO

for details. If you would like to use the second method, please note that the source for the export modules is in the `export/` subdirectory in the Tablix source tree.

Export modules are usually stored in the same directory as the fitness modules. While fitness modules can have arbitrary names, all export modules must have the prefix "export_" in their names. `tablix2_output` utility gets the name of the export module to load by concatenating "export_" and the name of the export format. For example, the following command line:

```
$ tablix2_output csv result0.xml
```

will cause the `tablix2_output` utility to load the `export_csv.so` export module.

2.3. Output extensions

2.3.1. Description

As you probably noticed, export function can only access the timetable in its basic chromosome form (fitness functions can also access slist and extension forms). It has already been mentioned that chromosome extension form resembles a human readable timetable format. Because of this it is often useful to use this form in the export function instead of the chromosome form.

However, the chromosome extension that is used in fitness functions usually isn't suitable for use in export modules because events (tuples) can be lost in the conversion from the chromosome form. Export modules therefore use another form called output extension that does not have this drawback.

The output extension outputext structure closely resembles the normal chromosome extension ext structure. It also defined by one constant and one variable resource type and also consists of a two-dimensional array. However instead of single tuple IDs in the normal extension the two-dimensional array now holds lists of tuples (stored in the tuplelist structure.

Consider the following example: if you construct an output extension *outputext* for a constant resource type with type ID *con_typeid* and a variable resource type with type ID *var_typeid*, then the following element in the two-dimensional array:

`outputext.list[`*c*`][`*v*`]`

is a pointer to the tuplelist structure which holds a list of tuple IDs of tuples (events) that are using both the constant resource with resource ID *c* and type ID *con_typeid* and the variable resource with resource ID *v* and type ID *var_typeid*.

> **Note:** If you don't understant this example, see the section on chromosome extensions in the first part of the HOW-TO (specially the part about visualization). Keep in mind that in most cases the variable resource type used in extensions is time.

The tuplelist structure holds a simple array of tuples. The field *tupleid* is an array of *tuplenum* tuple IDs.

2.3.2. Associated functions

Three functions are available for converting the chromosome form of the timetable to an output extension: `outputext_new()`, `outputext_update()` and `outputext_free()`. Following example demonstrates their use:

```
int export_function(table *tab, moduleoption *opt, char *file)
{
        outputext *ext;

        ext=outputext_new("dummy-constant-type", "dummy-variable-type");
        outputext_update(ext, tab);

        ...

        outputext_free(ext);

        return(0);
}
```

`outputext_new()` function allocates a new output extension structure and initializes its values. The first argument is the name of the constant resource type and the second argument is the name of the variable resource type. In case memory allocation fails or the requested resource types are not found, this function returns NULL. The example above doesn't do any error checking - a proper export module should report this condition as an error and about the execution.

`outputext_update()` function fills the two-dimensional array in the output extension with values from the timetable structure. After this function is called, the output extension is ready to use.

`outputext_free()` function frees any allocated memory taken by the output extension. It should be called after the extension is no longer needed to prevent memory leaks.

2.4. Internationalization

All character strings in the kernel structures are always UTF-8 encoded, even if the input XML file was in some other encoding. If your export format requires some other character encoding, you can use the libiconv library to transcode strings into other encodings.

If your exported format includes any messages that can be translated into other languages, please enclose them in a gettext translation macro like this:

```
fprintf(file, _("Hello world!"));
```

This way messages in your export module will be included in the Tablix translations. See GNU gettext documentation (http://www.gnu.org/software/gettext/manual/gettext.html) for more information.

Chapter 3. Example export modules

3.1. Comma separated values

Following is the source code of the `export_csv.so` export module. This is a simple export module that exports timetable data into a "comman separated values" (CSV) format that is suitable for import into other programs (for example spreadsheets). CSV format requires each line to be separated into multiple fields by the "," separator character. Fields containing strings must also be enclosed in double quotes. Fields with numbers are without quotes.

Output consists of four header lines and a table of events. First three lines contain the title, author and the address of the institution. Then comes a line that contains the fitness of the exported timetable and a header line for the table of events. Each event is represented as a single line. First field contains the name of the event and subsequent fields contain names of the resources used by this event, sorted by resource type.

```
#include "export.h"

int export_function(table *tab, moduleoption *opt, char *file)
{
        int typeid,tupleid;

        FILE *out;

        char *name;
        int resid;

        assert(tab!=NULL);

        if(file==NULL) {
                out=stdout;
        } else {
                out=fopen(file, "w");
                if(out==NULL) fatal(strerror(errno));
        }

        fprintf(out, "\"Title\",\"%s\"\n", dat_info.title);
        fprintf(out, "\"Address\",\"%s\"\n", dat_info.address);
        fprintf(out, "\"Author\",\"%s\"\n", dat_info.author);

        fprintf(out, "\"Fitness\",%d\n", tab->fitness);

        fprintf(out, "\"Event name\"");
        for(typeid=0;typeid<dat_typenum;typeid++) {
                fprintf(out, ",\"%s\"", dat_restype[typeid].type);
        }
        fprintf(out, "\n");

        assert(dat_typenum==tab->typenum);
```

```
        for(tupleid=0;tupleid<dat_tuplenum;tupleid++) {
                fprintf(out, "\"%s\"", dat_tuplemap[tupleid].name);

                for(typeid=0;typeid<dat_typenum;typeid++) {
                        assert(dat_tuplenum==tab->chr[typeid].gennum);

                        resid=tab->chr[typeid].gen[tupleid];
                        name=dat_restype[typeid].res[resid].name;

                        fprintf(out, ",\"%s\"", name);
                }

                fprintf(out, "\n");
        }

        if(out!=stdout) fclose(out);

        return 0;
}
```

As you can see, this export module consists of only the export function. A more complicated export module could contain additional functions called from the export function. The `export.h` header contains all prototypes and type definitions required in export modules.

Export function first opens the output file. As mentioned in the introduction, if the *`filename`* is NULL, then we write to the standard output instead of a file.

Following are four calls to the `fprintf()` function to write the header lines. Title of the exported timetable and information about the author can be found in the `dat_info` global variable (miscinfo structure). The fitness of the timetable can be found in the *`fitness`* field of the table structure.

Next we write the header of the event table. First column holds the name of the event, second column holds the resource of the first resource type, third column resource of the second resource type and so on. The first "for" loop therefore iterates through all resource types and prints out their names.

Following are two nested "for" loops that print out the whole event table. Outer loop iterates through all events (the current tuple ID is in the `tupleid` variable). We print the name of the event at the beginning of the line and enter the second loop. This loop again iterates through all resource types and gets the resource ID (`resid` variable) of the resource that the current event is using for each resource type. We then look up the name of this resource in the `dat_restype` global variable that holds pointers to all resource structures and print it.