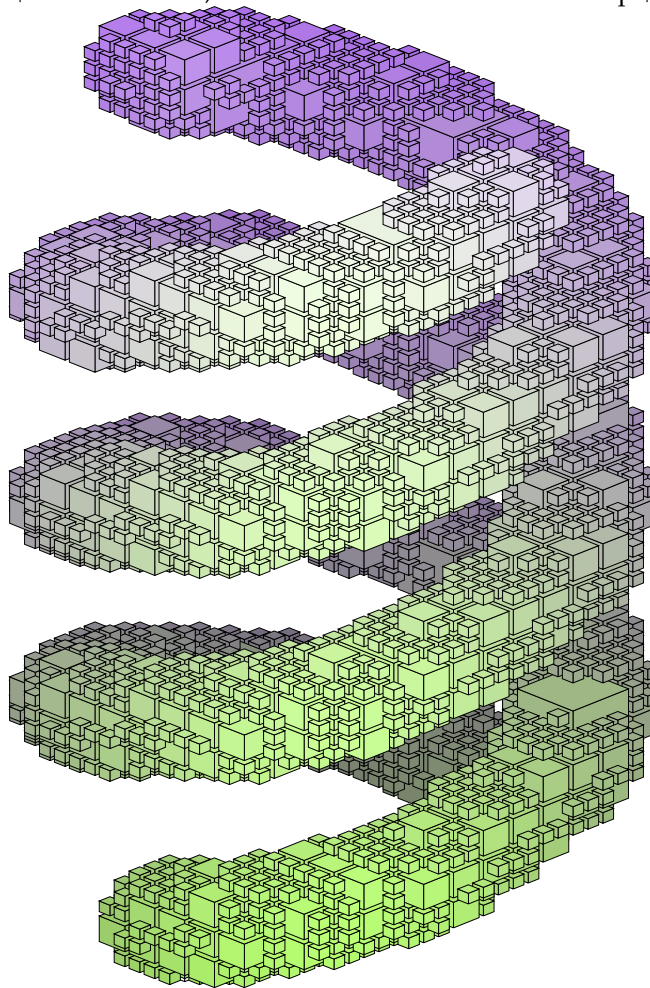


Tablix modules HOW-TO, part 1

Tomaž Šolc

\$Id: modules.db,v 1.20 2006-08-29 14:32:23 avian Exp \$



Tablix modules HOW-TO, part 1

Modules are pieces of code that are dynamically linked with the Tablix kernel at run time and provide most of the functionality of Tablix. This document describes in detail how to write and build new fitness modules. It explains how to use kernel API interfaces for this kind of modules, module documentation utilities and testing framework.

Copyright (C) 2005 by Tomaz Šolc.

Table of Contents

1. Introduction.....	1
1.1. Introduction to modules	1
1.2. Brief theoretical background on fitness functions.....	1
2. Kernel API	3
2.1. A basic module.....	3
2.1.1. Compiling your module.....	4
2.1.1.1. The simple way	4
2.1.1.2. The autotools way	5
2.1.2. Testing your module	5
2.2. Fitness function, chromosomes, event restriction	6
2.2.1. Module initialization	8
2.2.2. Event restriction	11
2.2.3. Fitness function	11
2.2.4. Testing the module.....	12
2.2.5. Possible improvements	12
2.3. Matrices, domains.....	13
2.3.1. About matrices	13
2.3.2. Modified module.....	14
2.3.3. Module initialization	15
2.3.4. Restriction handler.....	15
2.3.5. Testing the module.....	18
2.3.6. Discussion	18
2.4. Resource conflicts, slist, resource restriction.....	19
2.4.1. About slists.....	19
2.4.2. About conflicts.....	19
2.4.3. sametime.so module source code	20
2.4.4. Module initialization	22
2.4.5. Resource restriction handler.....	23
2.4.6. Fitness function	23
2.5. Timetable extensions	23
2.5.1. About extensions.....	23
2.5.2. holes.so module source code	26
2.5.3. Module initialization	28
2.5.4. Fitness function	28
2.5.5. Discussion	29
2.6. Dependent events	29
2.6.1. Some theory	29
2.6.2. About updater functions.....	30
2.6.3. Dependency solving	32
2.6.4. consecutive.so module.....	32
2.6.5. Registering updater functions.....	34
2.6.6. Updater function.....	35
2.6.7. Updater functions and resource domains	35
2.6.8. Porting modules from 0.3.1 to 0.3.2	36

2.7. Miscellaneous	37
2.7.1. Order of calling.....	37
3. Module documentation.....	38
3.1. Introduction	38
3.2. Comment block syntax	38
3.3. Module information block.....	39
3.4. Restriction and module option information blocks.....	40
4. Tablix Testing Framework	42
4.1. Introduction	42
4.2. How a test is processed.....	42
4.3. A basic test	42
4.4. How to write scheme code	44
4.5. A test case for the <code>fixed.so</code> module.....	45
4.6. A test case for the <code>consecutive.so</code> module.....	46

List of Tables

2-1. Example order of resources in a matrix (height = 4).....	13
---	----

Chapter 1. Introduction

1.1. Introduction to modules

As of version 0.0.4 Tablix supports loadable modules that include timetable fitness functions. The fitness 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. Fitness functions are then used by the genetic algorithm to select best timetables out of a large search space. 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. This document describes fitness modules although many things also apply to the export modules. Export modules specifics are detailed in the second part of this HOW-TO

I recommend reading the Tablix User's Manual and the Tablix Timetabling Model formal description 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 structures are also described in the text, but mostly only references to the reference manual will be given.

Note: All example XML configuration files and module source code can be found in the `examples/modules/` subdirectory in the Tablix source tree.

1.2. Brief theoretical background on fitness functions

Tablix uses a simple weighted sum to compute a fitness value of a timetable. This is implemented with a for loop in function `table_fitness()` in `modsup.c`. It looks like this in pseudo code:

```
FOR each partial fitness function defined by loaded modules DO
    CALL partial fitness function
    MULTIPLY the result with the weight of this module
    ADD the result to the total fitness of this timetable
DONE
```

Lower fitness value for a timetable means a better timetable. A perfect timetable that has no errors would have the fitness value equal to zero.

Each module usually defines one partial fitness function. This function in most cases returns the number of errors of a certain type in the timetable. There are some cases where this is not possible (for example when a module is calculating a dispersion of events). In that case the function should return lower values for better timetables. It is not mandatory that a timetable with fitness zero exists although it is desirable (a module can not be declared mandatory if its fitness function can never reach zero).

It is important that the return value of the partial fitness function depends only on one type of errors in the timetable. Because user can define which modules will be used by the kernel this enables her to customize the search for solution. By assigning different weight values to different modules she can assign more computational effort to solving a specific type of errors. This would not be possible if a fitness value depended on more than one type of errors. Experience also shows that partial fitness functions that are not orthogonal (if a value of one fitness function strongly affects the value of another) sometimes cause instability in the genetic algorithm.

Genetic algorithm gives best results if the fitness functions are continuous. This means that the fitness value should gradually decrease as the timetable gets better (has less errors). An example of the worst kind of a fitness function would be a function that returns zero for a perfect timetable and some fixed value for non-perfect ones

Chapter 2. Kernel API

2.1. A basic module

Let's begin with a simple "hello world" type of a module:

```
#include "module.h"

int module_init(moduleoption *opt)
{
    debug("Hello world!");
    debug("Weight: %d", option_int(opt, "weight"));

    return(0);
}
```

As you can see this module contains only the `module_init()` function. This function is called right after the kernel loads the module and is the only symbol that must be defined in all fitness modules for Tablix kernels in 0.2.x branch (prototype is defined in `modsup.h`, see `init_f`). No other function in the module is called unless you specifically request so in `module_init()`.

Note: `module_init()` is equivalent to `init_mod()` in 0.1.x kernels.

Parameter *opt* of `module_init()` is a structure holding all module options that were specified in the configuration file as `<option>` tags. You can get values of these options from the `moduleoption` structure by using following two functions:

```
char *option_str(moduleoption *opt, char *name)
```

This function returns the string contents of the option with the name *name* or returns NULL if the option was not specified in the XML file. You should free the returned string after use with `free()`.

```
int option_int(moduleoption *opt, char *name)
```

This function is equal to `option_str()`. Additionally it converts the contents of the option to an integer. If that is not possible or the option was not found, it returns `INT_MIN`.

Two options are always defined: The weight option always holds the integer value of the weight for this module that was specified in the XML configuration file. The mandatory option always contains either integer 1 or integer 0, depending whether the mandatory option for this module was set to "yes" or "no" in the configuration file.

Warning

Pointer to the linked list of module options is only passed to the `module_init()` function. Immediately after this function returns the memory used by the list is freed. This means that this pointer is only valid in `module_init()`.

This simple module also demonstrates the use of the error reporting functions that are defined in `error.h`. Functions `error()`, `info()` and `debug()` can be used to report various warnings and errors back to the user. They are used much like the `printf` function from the standard library.

You can also see that `module_init()` returns 0 in this case. This means that there were no errors. In case of errors this function should return a value less than zero. The same holds for all other functions we will later add to our fitness module. In fact that is the case with most functions in Tablix that return an integer.

You should avoid using `fatal()` in modules, because that function immediately terminates the kernel. Use `error()` to report the error and then use the function return value to signal that a fatal error has occurred in the module. The kernel will then terminate in a cleaner way.

2.1.1. Compiling your module

There are two ways of compiling your module. In most cases you would want to use the first way. The second way described here requires up to date autotools installed on your machine and is recommended only if you would like to compile your module in the exactly the same way the official modules in the distribution are compiled.

2.1.1.1. The simple way

Create a subdirectory in the Tablix source tree, for example `local/`. Name the file with the source code of your module `hello.c` and save it into the subdirectory you've made. Now create a file named `Makefile` in that directory with the following contents (you *must* use tabs to indent lines!):

```
CFLAGS = -Wall -O2
INCLUDES = -I../src -I..

MODULES = hello.so

all: $(MODULES)

%.so: %.o
    gcc -shared -Wl,-soname,$@ -o $@ $< -lc
%.o: %.c
    gcc $(CFLAGS) $(INCLUDES) -c -o $@ $<

clean:
    -rm *.o *.so

install:
    cp $(MODULES) /usr/local/lib/tablrx2
```

Later when you add more custom modules to your collection, you can add their names to the `MODULES =` line after `hello.so`. Just be sure to replace the `.c` extension with the `.so`.

Modules are usually installed in `/usr/local/lib/tablrx2`, but that may be different on your system, depending on any `--prefix` options during Tablrx configuration and / or your operating system. You can find the proper location by running the following command:

```
$ tablrx -v
```

Fix the path in the install rule if necessary.

You can now compile and install your module with the following command:

```
$ make
$ make install
```

2.1.1.2. The autotools way

Note: This method requires a recent version of Autoconf (<http://www.gnu.org/software/autoconf>), Automake (<http://www.gnu.org/software/automake>) and related software. This is the way the official modules in the distribution are compiled.

If you get any strange errors while running `autoconf` or `automake`, this 99% of cases means that you have the wrong version of autotools installed. Warnings about "underquoted definitions" are normal and can be ignored (they have nothing to do with Tablrx).

Put the file with the source code of your module into the `modules` subdirectory. Now you have to edit the `Makefile.am` file. First add the following two lines at the end of the file

```
hello_la_SOURCES = hello.c
hello_la_LDFLAGS = -module -avoid-version
```

Then add `hello.la` to the end of the line beginning with `pkglib_LTLIBRARIES`.

Now you have to run `autoconf` and `automake` from the top directory and rerun `./configure` to refresh the actual Makefiles. You can also run `make -f Makefile.cvs` if you are using the CVS version.

You can now compile and install your module with the following commands:

```
$ make
$ make install
```

2.1.2. Testing your module

First we need a Tablrx configuration file that uses our new module. Because this simple modulus doesn't do anything, we can keep it as simple as possible. We only need to define one resource and one event to keep the kernel happy.

```

<?xml version="1.0" encoding="iso-8859-1"?>
<ttm version="0.2.0">
  <modules>
    <module name="hello.so" weight="10" mandatory="yes"/>
  </modules>

  <resources>
    <variable>
      <resourcetype type="dummy-type">
        <resource name="dummy-resource"/>
      </resourcetype>
    </variable>
  </resources>

  <events>
    <event name="dummy-event" repeats="1"/>
  </events>
</ttm>

```

Save this configuration to a file named `hello.xml` and run Tablix. Because we used the `debug()` function in the module, you have to run Tablix with the `-d5` option to see the "Hello world!" message.

A few screenfulls of messages will scroll past in the console as you run Tablix. Because `hello.so` module hasn't defined any fitness functions all timetables get fitness value 0 and Tablix will only run for a few hundred generations before declaring it has found the solution. Somewhere at the beginning of the output you should find the "Hello world!" message.

```

$ tablix2 -d5 hello.xml
TABLIX version 0.2.1, PGA general timetable solver
Copyright (C) 2002-2005 Tomaz Solc

[tablix] PGA using 4 nodes on 1 host
[tablix] maximum 1 node will do local search
[tablix] multicasting XML data
[tablix] Initializing nodes
...
[xxxxx] xmlsup: variable dummy-type: 1 resources
[xxxxx] xmlsup: Loading module /home/avian/software/lib/tablix2/hello.so
[xxxxx] xmlsup: Hello world!
[xxxxx] xmlsup: Weight: 10
[xxxxx] kernel: I have 1 tuples
...

```

2.2. Fitness function, chromosomes, event restriction

Let's extend our basic module so that it will actually do something useful. Following is the source code of a module that will force selected event to use a certain resource from resource type

"dummy-type". It defines a new event restriction named "fixed-dummy-type". The contents of this restriction tell which resource from the "dummy-type" resource type should this event use.

```
#include <stdlib.h>
#include "module.h"

#define          RESTYPE          "dummy-type"

struct fixed {
    int tupleid;
    int resid;
};

static int fixed_num;
static struct fixed *fixed_tuples;

static resourcetype *dummy;

int handler(char *restriction, char *content, tupleinfo *tuple)
{
    int resid;

    resid=res_findid(dummy, content);
    if(resid==INT_MIN) {
        error(_("Resource '%s' not found"), content);
        return -1;
    }

    fixed_tuples[fixed_num].resid=resid;
    fixed_tuples[fixed_num].tupleid=tuple->tupleid;

    fixed_num++;

    return 0;
}

int fitness(chromo **c, ext **e, slist **s)
{
    chromo *dummy_c;
    int n, sum;
    int tupleid, resid;

    dummy_c=c[0];

    sum=0;

    for(n=0;n<fixed_num;n++) {
        tupleid=fixed_tuples[n].tupleid;
        resid=fixed_tuples[n].resid;

        if(dummy_c->gen[tupleid]!=resid) sum++;
    }
}
```

```

        return sum;
    }

int module_init(moduleoption *opt)
{
    fitnessfunc *f;

    dummy=restype_find(RESTYPE);
    if(dummy==NULL) {
        error(_("Resource type '%s' not found"), RESTYPE);
        return -1;
    }

    fixed_tuples=malloc(sizeof(*fixed_tuples)*dat_tuplenum);
    fixed_num=0;

    if(fixed_tuples==NULL) {
        error(_("Can't allocate memory"));
        return -1;
    }

    if(handler_tup_new("fixed-" RESTYPE, handler)==NULL) return -1;

    f=fitness_new("fixed",
        option_int(opt, "weight"),
        option_int(opt, "mandatory"),
        fitness);

    if(f==NULL) return -1;

    fitness_request_chromo(f, RESTYPE);

    return(0);
}

```

We have two new functions here: `fitness()` is our fitness function and `handler()` is the restriction handler for the type of event restrictions we defined.

Note: All global variables in modules should be declared as static.

2.2.1. Module initialization

Let's follow the `module_init()` function: First we try to find the `resourcetype` structure for the resource type named "dummy-type". If we can't find it, we report an error and exit. Note that this made our module dependend on the definition of a certain resource type in the configuration file.

Also note that all error messages are enclosed in gettext translation macros. You should use translation macro `_()` around any warning or error message that will be seen by users. Don't put it around resource type names or restriction types. Also don't use it with debug messages.

Next we allocate memory for the `fixed_tuples` array. This array stores the information about which tuples (events) had defined fixed resources defined by our restriction. Since it doesn't make sense to have more than one restriction of this type per event, we make the length of the array equal to the number of events (stored in the global `dat_tuplenum`). `fixed_num` stores the number of used restrictions.

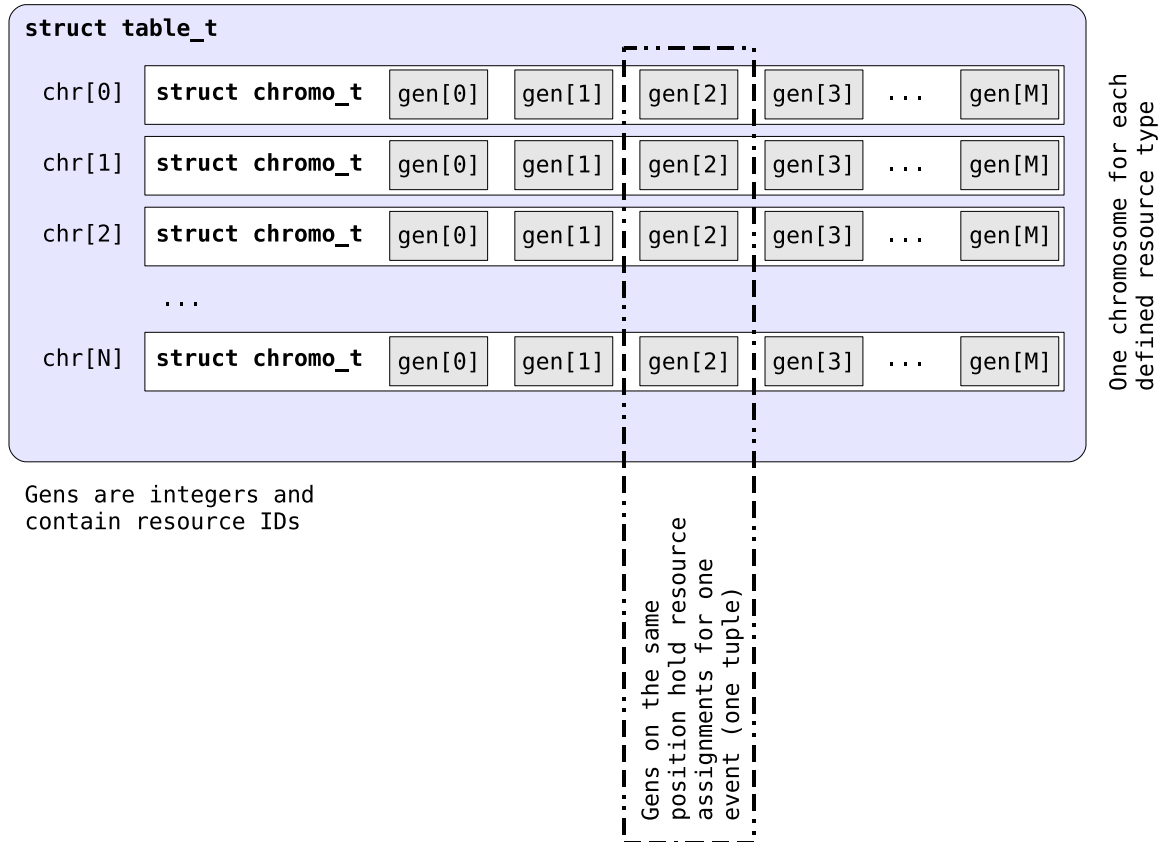
Now comes the important part. First we define a new event restriction type with the `handler_tup_new()`. Second we define our fitness function with the `fitness_new()`. We named this partial fitness value "fixed" (this name is used for example by `tablix2_plot` utility) and passed unchanged weight and mandatory values from the config file to `fitness_new()`.

Note: Your fitness functions should have short, descriptive names that should be easy for the user to connect with your module.

Finally we must tell the kernel in which form do we want the timetable to be passed to our fitness function. Kernel supports three forms of timetables: chromosome form, which is the native form of the genetic algorithm and is used in most cases, chromosome extension form and slist form. In this example we use a chromosome. Other two forms will be explained in later sections.

Figure 2-1. Timetable representation in the Tablix kernel.

N = dat_typenum (number of defined resource types)
M = dat_tuplenum (number of defined tuples)



A timetable is fully described with N chromosomes, where N is the number of resource types. For the purpose of this example a chromosome is an array of resource IDs. Length of the array is equal to the number of defined events. Mth resource ID in the array is the ID of the resource that is used by event with tuple ID equal to M.

Note: Resource ID is an unique numerical representation of a resource within its resource type (two resources can have the same resource ID if they are from different resource types).

Tuple ID is an unique numerical representation of an event. Two events always have different tuple IDs. Tuples defined with a single *<event>* tag and *repeats* greater than 1 have sequential IDs.

Caution

You may have noticed that the tupleinfo structure, passed by the kernel to event restriction handlers, also contains resource IDs for a tuple. You should never use those values in fitness functions. Resource IDs stored in tupleinfo are those defined in the XML file and may have changed in the timetable that must be evaluated by the fitness function.

Note: The length of the chromosome is stored in the *gennum* field of the chromosome structure. This number is equal to the number of defined events in the *dat_tuplenum* global variable. However you should use the *gennum* field when iterating through the whole chromosome in case of any future changes in the kernel.

In this example we request the chromosome for the resource type "dummy-type" to be passed to our fitness function by calling `fitness_request_chromo`.

Note: It does not matter if a resource type of the chromosome you requested is defined as variable or as constant in the configuration file.

2.2.2. Event restriction

The `handler()` function shouldn't require much explanation. It is called once for each event that had our restriction in the configuration file.

Note: If an event restriction is used in an `<event>` tag with *repeats* property greater than one, then the restriction handler function will be called multiple times, once for each event defined by the `<event>` tag (with *tuple* pointer set accordingly).

restriction argument holds the type of the restriction this handler was called for. Since we only used `handler()` function in one type of restrictions ("fixed-dummy-type") we don't need to check its value. Otherwise this argument would enable us to use a single function as a handler for multiple restriction types.

tuple is a pointer to *tupleinfo* structure that describes the event for which this restriction was defined. Here we only use it to get the tuple ID of the event and store its value in the *fixed_tuples* array. *content* holds the content of the restriction in string form. We interpret this string as a name of a resource and try to find and store its resource ID by using the *res_findid* function.

2.2.3. Fitness function

`fitness()` function is more interesting. Kernel passes to it three arrays with the requested forms of the timetable to evaluate. First is an array of pointers to chromosomes *c*. First pointer points to the chromosome for the first requested resource type, second pointer to the second requested resource type, etc. *e* and *s* are similar arrays of pointers to chromosome extensions and *slists*. Since we only requested one chromosome in this example a pointer to it is stored in the first location in the chromosome array.

The rest of fitness function is a loop that checks each tuple in the *fixed_tuples* array if it is using the correct resource. If it is not the error count *sum* is increased by one. When all tuples have been checked the function exits, returning the total number of tuples that are not using the correct resource. Fitness functions must always return a positive integer.

Note: Fitness function should not change any structures or arrays that are passed to it.

Tip: Fitness functions are one of the most often called functions in Tablix. It must be called for each timetable in a generation. One generation includes by default 500 timetables and it takes many thousand generations to find a solution. Because of this they should be as optimized as possible.

2.2.4. Testing the module

We need to construct an XML configuration file that will use the event restriction we defined with our module. We also have to define a few resources of the type "dummy-type" because our module depends on this resource type. To keep things simple, only one event is defined in the following example.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<ttm version="0.2.0">
  <modules>
    <module name="fixed.so" weight="10" mandatory="yes"/>
  </modules>

  <resources>
    <variable>
      <resourcetype type="dummy-type">
        <linear name="#" from="1" to="50"/>
      </resourcetype>
    </variable>
  </resources>

  <events>
    <event name="dummy-event" repeats="1">
      <restriction type="fixed-dummy-type">30</restriction>
    </event>
  </events>
</ttm>
```

Again save this file under a name `fixed.xml` and run Tablix. You will have to wait a few moments for Tablix to find a solution. You can then examine one of the result files:

```
<event name="dummy-event" repeats="1" tupleid="0">
  <restriction type="fixed-dummy-type">30</restriction>
  <resource type="dummy-type" name="30"/>
</event>
```

You can see that Tablix scheduled event "dummy-event" so, that it is using resource with the name "30" as we requested (resource "30" is one of 50 resources with names from "1" to "50" that were defined with the `<linear>` tag).

2.2.5. Possible improvements

This module can be improved in many ways. First, it does not check if there is more than one restriction per event in the configuration file (in that case the number of errors could never reach zero). It could also cause a segmentation fault in that case because the number of defined restrictions would exceed the number of defined events. Second, it depends on the definition of the "dummy-type" resource type in the configuration file while such dependency isn't strictly necessary (this module is general enough that it would be useful in a number of different situations). The solution to these problems is left as an exercise for the reader.

2.3. Matrices, domains

2.3.1. About matrices

Sometimes it is useful if resources of one resource type can be arranged in a two dimensional array. The most common example of this is when you have a fixed number of time slots per day and a fixed number of days per week. Such resource types are called matrix resource types or matrices.

Note: Matrix resource types are a generalization of days / periods in 0.1.x Tablix kernels. New kernel allows any resource type to be a matrix resource type. Many modules that were ported from the older kernels still depend on the time resource type to be a matrix.

All resources in a matrix always have special names in the form of two integers separated by a single space. First integer gives the column index (x coordinate) and the second integer gives the row index (y coordinate). Resources in a matrix are also always ordered first by rows and then by columns (see `res_new_matrix()` function documentation for more information).

Table 2-1. Example order of resources in a matrix (height = 4)

Resource ID	Resource name
0	"0 0"
1	"0 1"
2	"0 2"
3	"0 3"
4	"1 0"
5	"1 1"
6	"1 2"
7	"1 3"
8	"2 0"
...	...

Note: Resource type is a matrix only and only if all of its resources are defined by a single `<matrix>` tag. It is valid to combine `<matrix>` tag with other tags for the definition of resources, but the resulting resource type is not a matrix.

2.3.2. Modified module

Let's say we want to modify the previous example module so that we could define only the row of a resource an event will be using, not the exact resource. We will also modify the module so that it will be using resource domains, a new feature in 0.2.x kernels.

```
#include <stdio.h>
#include <stdlib.h>
#include "module.h"

#define          RESTYPE          "dummy-type"

static resourcetype *dummy;
static int width, height;

int handler(char *restriction, char *content, tupleinfo *tuple)
{
    int row;
    int result;
    int typeid;

    int *resid_list;
    int resid_num;

    int n;

    domain *dom;

    result=sscanf(content, "%d", &row);
    if(result!=1) {
        error(_("Row index must be an integer"));
        return -1;
    }

    if(row<0||row>height-1) {
        error(_("Row index must be between 0 and %d"), height-1);
        return -1;
    }

    typeid=dummy->typeid;

    resid_list=malloc(sizeof(*resid_list)*width);
    if(resid_list==NULL) {
        error(_("Can't allocate memory"));
        return -1;
    }
}
```

```

    }

    for(n=0;n<width;n++) resid_list[n]=row+n*height;
    resid_num=width;

    dom=tuple->dom[typeid];

    domain_and(dom, resid_list, resid_num);

    free(resid_list);

    return 0;
}

int module_init(moduleoption *opt)
{
    int result;

    dummy=restype_find(RESTYPE);
    if(dummy==NULL) {
        error(_("Resource type '%s' not found"), RESTYPE);
        return -1;
    }

    result=res_get_matrix(dummy, &width, &height);
    if(result) {
        error(_("Resource type " RESTYPE " is not a matrix"));
        return -1;
    }

    if(handler_tup_new("row-" RESTYPE, handler)==NULL) return -1;

    return(0);
}

```

2.3.3. Module initialization

First thing you might notice is that this module doesn't define a fitness function. Because of this module initialization function is pretty straightforward. First we try to find our resource type. Then we use the `res_get_matrix()` function to get the dimensions `width` and `height` of the matrix. It is important that we check whether the resource type is really a matrix: `res_get_matrix()` will return -1 if it cannot determine the dimensions and 0 on success. The last thing in the initialization is the familiar call to `handler_tup_new()` to define a new tuple restriction handler.

2.3.4. Restriction handler

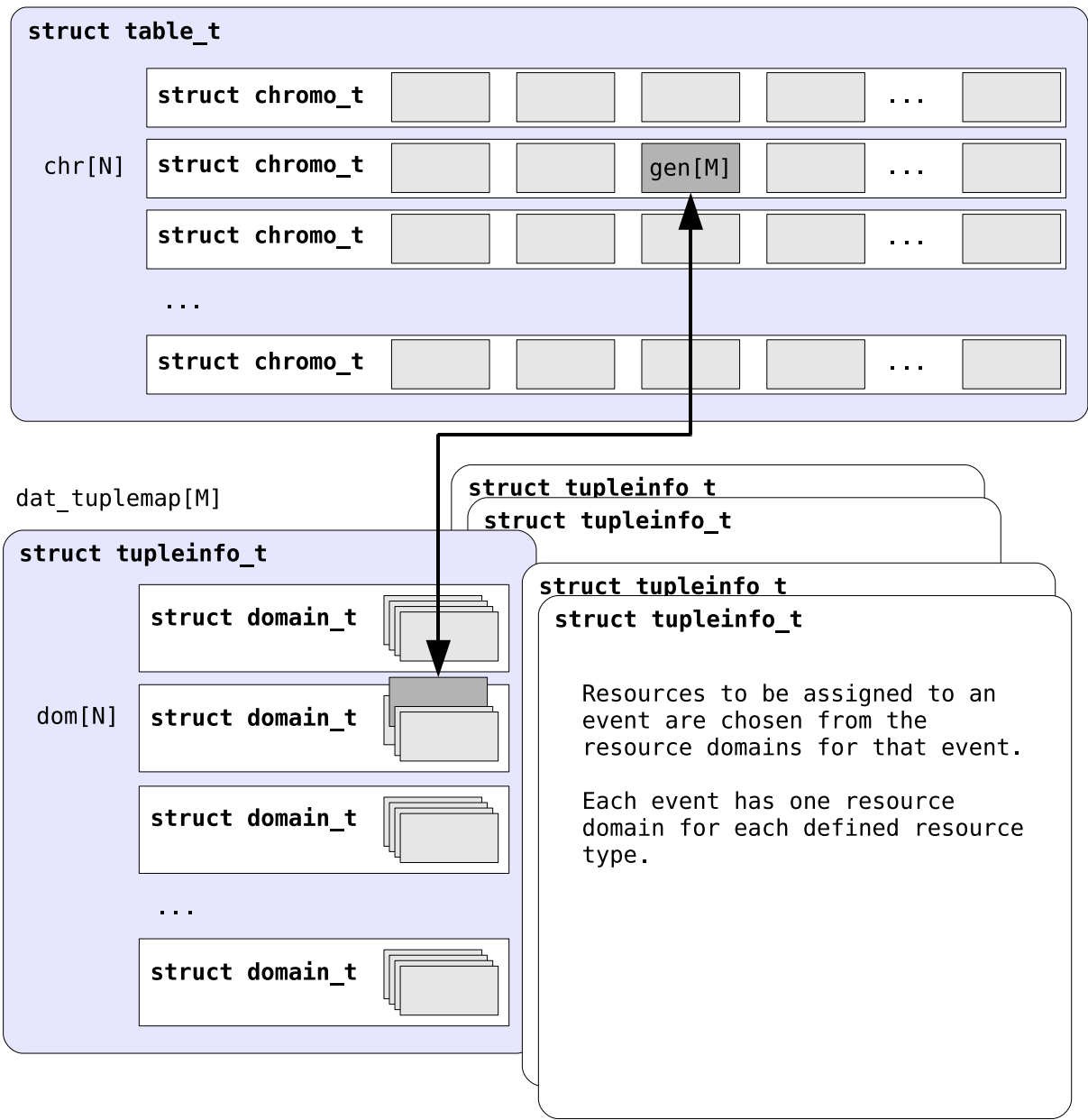
As you can see the important part of the code is this time in the restriction handler. Instead of using a fitness function to tell the genetic algorithm which timetables are better we will simply narrow the search space of the genetic algorithm. This of course means that it is impossible to make a non-mandatory restriction with this approach. As far as the genetic algorithm is concerned, timetables that do not satisfy our restriction are not even considered for a solution.

If you look at the restriction handler code you can see that we first read the row number specified by the user and do a check if it falls within the allowed range. Then we make a list of all resource IDs our event can use. Since we know in which row the allowed resources must be and we know how resources are ordered in a matrix it is quite straightforward to construct it.

Array `resid_list` holds the resource ID values and integer `resid_num` holds the length of the array. There are exactly `width` resources in a row, so we can set the length immediately. Resources are ordered first by row numbers and then by column numbers. So the first resource in the row `row` has the ID equal to `row`. The distance to the second resource is exactly `height` resources.

Now we pass the list of allowed resource to the `domain_and()` function. This function takes a resource domain and a list of resources as its arguments. Then it removes all resources from the domain that aren't also in the list. Note that this effectively narrows the search space to the lowest common denominator of all restrictions of this type, even those defined by other modules using `domain_and()` function.

Figure 2-2. Resource domains in Tablix kernel.



Note: A resource domain is a list of resources that can be used by an event. Each event has one resource domain for each defined resource type.

Pointers to domain structures for the tuple the restriction handler was called for can be found in the tupleinfo structure. The `dom` structure field contains an array of pointers to domain structures, ordered by resource type ID. Because of this we must first obtain the resource type ID for our

"dummy-type" resource type. Since we already have a pointer to that type, we can simply read the type ID from the *type* field.

Note: If you need to adjust resource domains outside of a tuple restriction handler, you should use the pointers to domain structures provided in the *dat_tuplemap* array.

2.3.5. Testing the module

Again we can construct a simple test case to check if the module is working correctly:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<ttm version="0.2.0">
  <modules>
    <module name="row.so" weight="10" mandatory="yes"/>
  </modules>

  <resources>
    <variable>
      <resourcetype type="dummy-type">
        <matrix width="10" height="10"/>
      </resourcetype>
    </variable>
  </resources>

  <events>
    <event name="dummy-event" repeats="1">
      <restriction type="row-dummy-type">3</restriction>
    </event>
  </events>
</ttm>
```

If you run Tablix on it and check one of the result files you will get a result similar to this one:

```
<event name="dummy-event" repeats="1" tupleid="0">
  <restriction type="row-dummy-type">3</restriction>
  <resource type="dummy-type" name="4 3"/>
</event>
```

You can see that the row of the chosen resource (second number in the resource name) is equal to the requested row of resources. If you run Tablix a couple of times with this same configuration file and compare the results you will see that the column of the resource is chosen randomly while the row will always stay the same.

2.3.6. Discussion

You should always use resource domains in your module if it is at all possible. Because this way you are limiting the search space Tablix will find the solution faster if there are more restrictions like this. On the other hand if there are more restrictions that are using fitness functions Tablix will need more time because it takes time to evaluate timetables with more fitness functions.

Also Tablix will detect if a domain does not contain any resources. This means that the user has applied an impossible combination of restrictions to an event and that a solution can not be found. This is one of the few cases where Tablix can predict that a solution to the problem described in the configuration file does not exist. On the other hand an impossible combination of restrictions that are using fitness functions is in most cases impossible to detect automatically.

2.4. Resource conflicts, slist, resource restriction

2.4.1. About slists

Slist is a special representation of the timetable that can be requested by `module_init()` function. In that case it is calculated by the kernel and passed to that module's fitness function.

Often a fitness function must calculate a list of events that are using a certain resource of a certain resource type (the most common example is to check if two events are using the same resource). This check can be very time consuming if performed on a chromosome form of the timetable but can be performed an order of the magnitude faster if you first translate the timetable from a chromosome form to a slist form. Because it would be a waste CPU cycles and memory if each module would preform this translation by itself, the kernel calculates a slist only once and then passes a pointer to all modules that requested this slist.

If you take a look at the API reference manual, you will see that a slist structure consists of *varnum* arrays, each array holding zero or more tuple IDs. *varnum* is equal to the number of resource in the chosen resource type (its ID is stored in *var_typeid*). The Nth array holds the tuple IDs of all events that are using the resource with resource ID equal to N and resource type ID equal to *var_typeid*.

Note: As you can see, the number of possible slists that can be generated for a certain timetable is equal to the number of defined variable resource types in that timetabling problem. In practice only one or two slists need to be generated. Also note that it is possible to generate a slist for a constant resource type, however there are no practical uses for such slists.

Unlike chromosomes, slists take some time to generate. However once a slist is generated (for a certain resource type) then almost no additional processor time is required to pass the pointer to the generated slist to two or more modules. So, use a slist if you know that this same slist will be requested by another module used by the timetabling problem and that its use will result in a small speed-up of your fitness function. On the other hand if the use of the slist will result in a major speed-up (for example from $O(n^2)$ to $O(n)$ time) then use the slist in any case.

2.4.2. About conflicts

The use of conflicts somewhat depends on the agreement of authors of a certain group of modules. The kernel merely provides a mechanism of tagging which resource conflicts with which by exporting two functions: `res_set_conflict()` and `res_get_conflict()`. The only part of the Tablix kernel that uses this information is the generation of timetable extensions.

Note: A resource always conflicts with itself. By default no resources conflict. Modules can then only set conflicts between resources and can not unset them.

For example in school scheduling conflicts are used with constant resource types (in this case groups of students and teachers). Two events that are attended by two conflicting groups of students can never be scheduled at two places at the same time. Same with two conflicting teachers.

The kernel treats conflicts asymmetrically. This means that resource 1 can conflict with resource 2 while at the same time resource 2 does not conflict with resource 1. Most timetabling problems however need symmetrical conflicts (for example school scheduling). This means that when setting a conflict between two resources, you must call `res_set_conflict()` twice (second time with reversed order of the arguments).

Conflicts are treated as not transitive in the kernel, which means that if resource 1 conflicts with resource 2 and resource 2 conflicts with resource 3, this does not mean that resource 1 conflicts with resource 3. Some modules can however treat conflicts are transitive.

2.4.3. sametime.so module source code

Following is the important part of the `sametime.so` module source code (full source is available in the distribution). You should be familiar with most of the function calls in it by now.

If you are not familiar with the function of this module (it is one of the standard modules used in school scheduling) have a look at the Module reference documentation.

```
int getconflict(char *restriction, char *cont, resource *res1)
{
    resource *res2;

    res2=res_find(res1->restype, cont);

    if(res2==NULL) {
        error(_("invalid resource in conflicts-with restriction"));
        error(_("resource: %s resource type: %s"), cont, res1->restype);
        return(-1);
    }

    res_set_conflict(res1, res2);
    res_set_conflict(res2, res1);

    return 0;
}
```

```

int module_fitness(chromo **c, ext **e, slist **s)
{
    int a,b,m;
    int n;
    int sum;
    slist *list;
    chromo *time, *room, *class, *teacher;
    resourcetype *teacher_type, *class_type;

    list=s[0];

    room=c[0];
    time=c[1];
    class=c[2];
    teacher=c[3];

    teacher_type=teacher->restype;
    class_type=class->restype;

    sum=0;
    for(m=0;m<time->gennum;m++) {
        a=time->gen[m];

        for(n=0;n<list->tuplenum[a];n++) if(list->tupleid[a][n]<m) {
            b=list->tupleid[a][n];
            if (room->gen[m]!=room->gen[b]) {
                if(res_get_conflict(teacher_type,
                                    teacher->gen[m],
                                    teacher->gen[b])) {
                    sum++;
                }
                if(res_get_conflict(class_type,
                                    class->gen[m],
                                    class->gen[b])) {
                    sum++;
                }
            }
        }
    }

    return(sum);
}

int module_precalc(moduleoption *opt)
{
    ...
}

int module_init(moduleoption *opt)
{
    fitnessfunc *fitness;

```

```

handler_res_new("class", "conflicts-with", getconflict);
handler_res_new("teacher", "conflicts-with", getconflict);

precalc_new(module_precalc);

fitness=fitness_new("same time",
                    option_int(opt, "weight"),
                    option_int(opt, "mandatory"),
                    module_fitness);
if(fitness==NULL) return -1;

if(fitness_request_chromo(fitness, "room")) return -1;
if(fitness_request_chromo(fitness, "time")) return -1;
if(fitness_request_chromo(fitness, "class")) return -1;
if(fitness_request_chromo(fitness, "teacher")) return -1;

fitness_request_slist(fitness, "time");

return(0);
}

```

2.4.4. Module initialization

There are three function calls in the `module_init()` function that we haven't discussed yet. First one is the call to `handler_res_new()`, which registers a new resource restriction handler in the kernel. It can be used in much the same way as `handler_tup_new()`: first argument is the name of the resource type to which this restriction can be applied. Second argument is the name of the restriction and the third argument is a pointer to the handler function.

Note: You can pass a NULL pointer instead of the resource type name in the first argument of the `handler_res_new()` function. In that case the restriction will be registered for all resource types.

The second new function call is `precalc_new()`. This simply registers a function in the module that will be called after all restriction handlers. This function can be used for example to perform a sanity check on any data structures used by the module and give a warning to the user if the problem looks unsolvable. It can also be used to precalculate any look-up tables that may be needed by the fitness function (since look-up tables are calculated only once, it often makes sense to move as much code from the fitness function to the precalculate function).

Note: The prototype of the precalculate function is exactly the same as that of the `module_init()`. However the module option `*opt` argument is always NULL. Pointer to the module options linked list is only passed to the `module_init()` function.

Precalculate functions obey the same rule as most other functions in Tablix: return 0 if successful, or -1 on error.

Finally there is a call to `fitness_request_slist()` function that requests a slist. The first argument is a pointer to the fitness function structure and the second argument is the name of the variable resource type for which the slist should be generated. If you request more than one slist for a single fitness function then slists will be passed to it in the same order as the `fitness_request_slist()` functions were called.

2.4.5. Resource restriction handler

The `getconflict()` function is quite simple. It gathers two resource IDs: ID of the resource it was called for (pointer `res1` points to its resource structure) and the ID of the resource whose name was passed to it as an argument. Then it uses the `res_set_conflict()` function two times to tag these two resources as conflicting.

2.4.6. Fitness function

This fitness function checks whether two conflicting teachers or two conflicting groups of students are scheduled to have a lecture (event) at the same time. Since a resource always conflicts with itself, this also means that the same fitness function checks whether a group or a teacher must be at the same time at two different lectures.

First, we store all pointers to requested chromosomes and a slist to local variables. Since only one slist was requested the pointer to it is in the first place in the `s` array.

The first loop goes through all defined events (current tuple ID is stored in `m`). For each event we get the time slot it is using and store its resource ID in `a`.

The second loop iterates through the `ath` array in the slist. We requested a slist for the "time" resource type, so this means that this for-loop iterates through all tuple IDs of events that are using the same time slot as the event in `m`. The tuple ID of the second event is stored in `b`. The if conditional is required to skip those event comparisons that were already made (without it all events would be compared twice).

The code in the inner loop then increments the error counter `sum` if two conflicting teachers or student groups are scheduled at different rooms at the same time.

`res_get_conflict()` is a macro. The first argument must be a pointer to the resource type structure. The second and the third argument are resource IDs of resources that should be checked for a conflict. The macro evaluates to 1 if the first resource conflicts with the second or 0 otherwise.

2.5. Timetable extensions

2.5.1. About extensions

Timetable extension is the third form of the timetable that can be passed to the fitness function. Its structure resembles a human-readable form of the timetable, like for example the output of the

htmlcss export module. Because of that it is often the most convenient form to check for errors in timetables for human resources.

If you take another look at the API reference manual you can see that a timetable extension is defined by one constant resource type and one variable resource type (stored in *con_typeid* and *var_typeid* respectively). In comparison, *slist* is defined by only one variable resource type. Timetable data is stored in a two dimensional array of tuple IDs in the *tupleid* field. The first dimension goes from 0 to N-1, where N is the number of defined resources of the variable resource type *var_typeid* and the second dimension goes from 0 to M-1, where M is the number of defined resources of the constant resource type *con_typeid*.

If an event is using a resource n of type *var_typeid* and a resource m of type *con_typeid* at the same time, then its tuple ID will be stored in the *e->tupleid[n][m]*, where *e* is a pointer to the extension struct. If no event is using this combination of resources, then the value stored in the array will be -1.

To better visualize this array of tuple IDs consider the following example from school scheduling. Let's say that we choose teachers as the constant resource type and time slots as the variable resource type. Also, since we are looking at a school timetable, time slots are in a form of a matrix with five days and three time slots per day.

Tuple IDs stored in the array represent events (in this case lectures) that are scheduled for this teacher at the specified time. If no event is scheduled for a certain teacher at a certain time slot then the value in the array will be -1.

Figure 2-3. Visualization of a timetable extension in school scheduling.

First teacher (resource ID = 0)					
	Mon	Tue	Wed	Thu	Fri
0	e->tupleid[0][0]	e->tupleid[3][0]	e->tupleid[6][0]	e->tupleid[9][0]	e->tupleid[12][0]
1	e->tupleid[1][0]	e->tupleid[4][0]	e->tupleid[7][0]	e->tupleid[10][0]	e->tupleid[13][0]
2	e->tupleid[2][0]	e->tupleid[5][0]	e->tupleid[8][0]	e->tupleid[11][0]	e->tupleid[14][0]

Second teacher (resource ID = 1)					
	Mon	Tue	Wed	Thu	Fri
0	e->tupleid[0][1]	e->tupleid[3][1]	e->tupleid[6][1]	e->tupleid[9][1]	e->tupleid[12][1]
1	e->tupleid[1][1]	e->tupleid[4][1]	e->tupleid[7][1]	e->tupleid[10][1]	e->tupleid[13][1]
2	e->tupleid[2][1]	e->tupleid[5][1]	e->tupleid[8][1]	e->tupleid[11][1]	e->tupleid[14][1]

Third teacher (resource ID = 2)					
	Mon	Tue	Wed	Thu	Fri
0	e->tupleid[0][2]	e->tupleid[3][2]	e->tupleid[6][2]	e->tupleid[9][2]	e->tupleid[12][2]
1	e->tupleid[1][2]	e->tupleid[4][2]	e->tupleid[7][2]	e->tupleid[10][2]	e->tupleid[13][2]
2	e->tupleid[2][2]	e->tupleid[5][2]	e->tupleid[8][2]	e->tupleid[11][2]	e->tupleid[14][2]

It should be obvious now that an extension only represents a whole timetable if there is a one-to-one mapping of events to constant-resource/variable-resource pairs (i.e. no two events are using the same combination of the constant resource and variable resource of the chosen types). If this is not true than some events lost when converting a timetable from the chromosome format to the extension since only one tuple ID can be stored at one position in the *tupleid* array. In this case a random event will be chosen from the group of conflicting events and its tuple ID will be stored in the array.

Note: This means that you won't always find all defined events in an extension. Some events may be lost, because they are using the same pair of resource than some other events.

Because of this limitation almost all modules that are using timetable extensions only work correctly when they are used together with a module (set to mandatory) that prevents this loss of events from happening. In school scheduling this is the role of the *sametime.so* module.

It was mentioned above that extensions are the only part of the kernel that is using the information about resource conflicts. Conflicting constant resources are handled as a single resource in this case. Conflicts in variable resources are ignored. This means that if a constant resource *m1* conflicts with a constant resource *m2* then events for both *m1* and *m2* will show in *e->tupleid[x][m1]*. On the

other hand only events for m2 will show in `tupleid[x][m2]` (kernel treats conflicts asymmetrically, remember?).

2.5.2. holes.so module source code

`holes.so` fitness module is used in school scheduling and similar timetabling problems and searches for so called holes in the timetable. A hole in a timetable for a certain constant resource is defined as one or more free time slots in a day (time slots that aren't used by any event that uses this constant resource) surrounded on both sides (before and after on the same day) with non-free time slots.

Example 1: Timetable with three lectures in the middle of the day is considered without holes.

Example 2: Timetable with one lecture in the morning, then a pause for two time slots and then two lectures is considered to have one hole that is two time slots wide.

Holes are in most cases annoying since they mean that for example a teacher or a group of students must wait for an hour between lectures. In some cases they are even forbidden by various regulations.

Following is an excerpt of the `holes.so` module. Full source code is included in the distribution.

```
static int periods, days;

int fitness(chromo **c, ext **e, slist **s)
{
    int first,last;
    int free,nonfree;
    int sum;

    ext *timext;
    int connum, con_resid;
    int day, period, var_resid;

    timext=e[0];

    connum=timext->connum;

    sum=0;

    for(con_resid=0;con_resid<connum;con_resid++) {
        var_resid=0;
        for(day=0;day<days;day++) {
            first=-1;
            last=-1;
            free=0;
            nonfree=0;

            for(period=0;period<periods;period++) {
                if(timext->tupleid[var_resid][con_resid]==-1) {
                    free++;
                } else {
                    nonfree++;
                }
            }
        }
    }
}
```

```

        last=period;
        if (first==-1) first=period;
    }
    var_resid++;
}

    if (last!=-1) {
        sum=sum+(periods-nonfree-first-(periods-1-last));
    }
}

};
return(sum);
}

int module_init(moduleoption *opt)
{
    fitnessfunc *f;
    moduleoption *result;

    char *type;
    char fitnessname[256];
    int n;

    resourcetype *time;

    time=restype_find("time");
    if(time==NULL) {
        error_(_("Resource type '%s' not found"), "time");
        return -1;
    }

    n=res_get_matrix(time, &days, &periods);
    if(n) {
        error_(_("Resource type %s is not a matrix"), "time");
        return -1;
    }

    result=option_find(opt, "resourcetype");
    if(result==NULL) {
        error_(_("module '%s' has been loaded, but not used"), "holes.so");
    }
    while(result!=NULL) {
        type=result->content_s;

        snprintf(fitnessname, 256, "holes-%s", type);

        f=fitness_new(fitnessname,
            option_int(opt, "weight"),
            option_int(opt, "mandatory"),
            fitness);

        if(f==NULL) return -1;
    }
}

```



```

        n=fitness_request_ext(f, type, "time");
        if(n) return -1;

        result=option_find(result->next, "resourcetype");
    }

    return(0);
}

```

2.5.3. Module initialization

Since this module is intended for use in school scheduling, we expect the "time" resource type to be a matrix (width of the matrix is equal to the number of days in a week and height is equal to the number of time slots per day). We store the dimensions in `days` and `periods` global variables (we will use them later in the fitness function).

The "while" loop that follows iterates through all module options with name "resourcetype". With this option user can specify constant resource types that will have their timetables checked for holes (for example teacher, classes, etc.)

First we try to find the pointer to the first option by using the `option_find()` function (`option_str()` function can't be used if you expect more than one option with the same name - see API documentation). If we can't find even one option with this name (`option_find()` returns NULL), then this module isn't affecting the timetable solution and we can report a warning.

From the standpoint of the kernel we define a new fitness function for each such option. In reality we define a single `fitness()` function multiple times. Each time with a new name that includes the name of the resource type and each time we request a different timetable extension with `fitness_request_ext()`. This means that for each timetable evaluation our `fitness()` function will get called one or more times, depending on the number of "resourcetype" options the user supplied.

`fitness_request_ext()` is used in much the same way as `fitness_request_slist()` or `fitness_request_chromo()`. First argument must be a pointer to the fitness function structure, the second and the third argument are the names of the constant and the variable resource types respectively. Here, we request an extension with a constant resource type supplied by the user and "time" variable resource type.

2.5.4. Fitness function

Fitness function of this module is a bit more complicated than the previous ones. Our requested extension is stored in `e` and the number of defined resource of the chosen constant resource type is stored in `connum` (we obtain this number from the extension structure, since we don't know for which constant resource type this instance of `fitness()` was called for).

The outer most loop iterates `con_resid` through IDs of all defined constant resources. Since holes in the timetable are determined on a day basis, we also have a second loop that iterates through all days. `var_resid` holds resource ID of the current time slot.

The inner most loop iterates through all time slots in a day and determines: 1) period (y coordinate in the matrix) of the first non-free time slot `first` on this day, 2) period of the last non-free time slot `last` and 3) number of non-free time slots on this day.

If there are no non-free time slots (`last` is equal to -1), then by definition there are also no holes in the timetable for this day. If there are non-free time slots then the number of holes can be calculated by a simple formula: number of time slots between the `first` and the `last` time slot minus the number of non-free time slots equals the number of holes. The calculated number of holes for the current day is added to the error sum `sum`

Note: After the `period` loop finishes, the `var_resid` has advanced for exactly `periods` resource IDs. Because of the specific order of resources in a resource matrix this means that `var_resid` is then pointing to the first time slot on the next day. After the `day` loop finishes, the `var_resid` has advanced `periods*days` which is exactly the number of time slots available in the timetable.

2.5.5. Discussion

Timetable extensions are very computationally expensive. In most cases they are more expensive than `slists`. The discussion about `slists` is therefore also applicable for extensions.

The time required to calculate the extension depends on the number of defined constant and variable resources. It also depends on the use of conflicts in the specified constant resource type. If the timetable has non-trivial conflicts defined in the constant resource type (if any resource conflicts with a resource other than itself) then extensions take longer to compute.

2.6. Dependent events

2.6.1. Some theory

Most timetable constraints can be satisfied in more than one way. Imagine for a moment a timetabling problem that uses only the `sametime.so` module with the `mandatory` option set. This problem can have literally millions of solutions that all satisfy the constraint set by the `sametime.so` module. But on the other hand the number of all combinations of such a timetable (i.e. the space that must be searched to find one of those solutions) is million or billion times greater than the number of solutions. There is also no trivial way to find a solution short of checking all possible combinations.

Timetabling problem described is a perfect example of a problem that is ideal for solving with a genetic algorithm. Most timetabling constraints fall into that category, however sometimes we come across a constraint that is an exact opposite: it has only one or at least very few solutions and the solution is trivial if we know the nature of the problem.

It is very hard for a genetic algorithm to find a solution to such a problem since the only knowledge the algorithm has about the nature of the problem is the shape of the fitness function. This means

that the fitness function must be very carefully chosen so that it "guides" the algorithm to the single solution.

However it would be wasteful to use the complicated genetic algorithm to solve a problem that has a trivial solution. Because of this Tablix supports updater functions since the 0.3.1 release. These functions basically enable the module author to mix her own algorithm with the genetic algorithm that is running in the background.

Warning

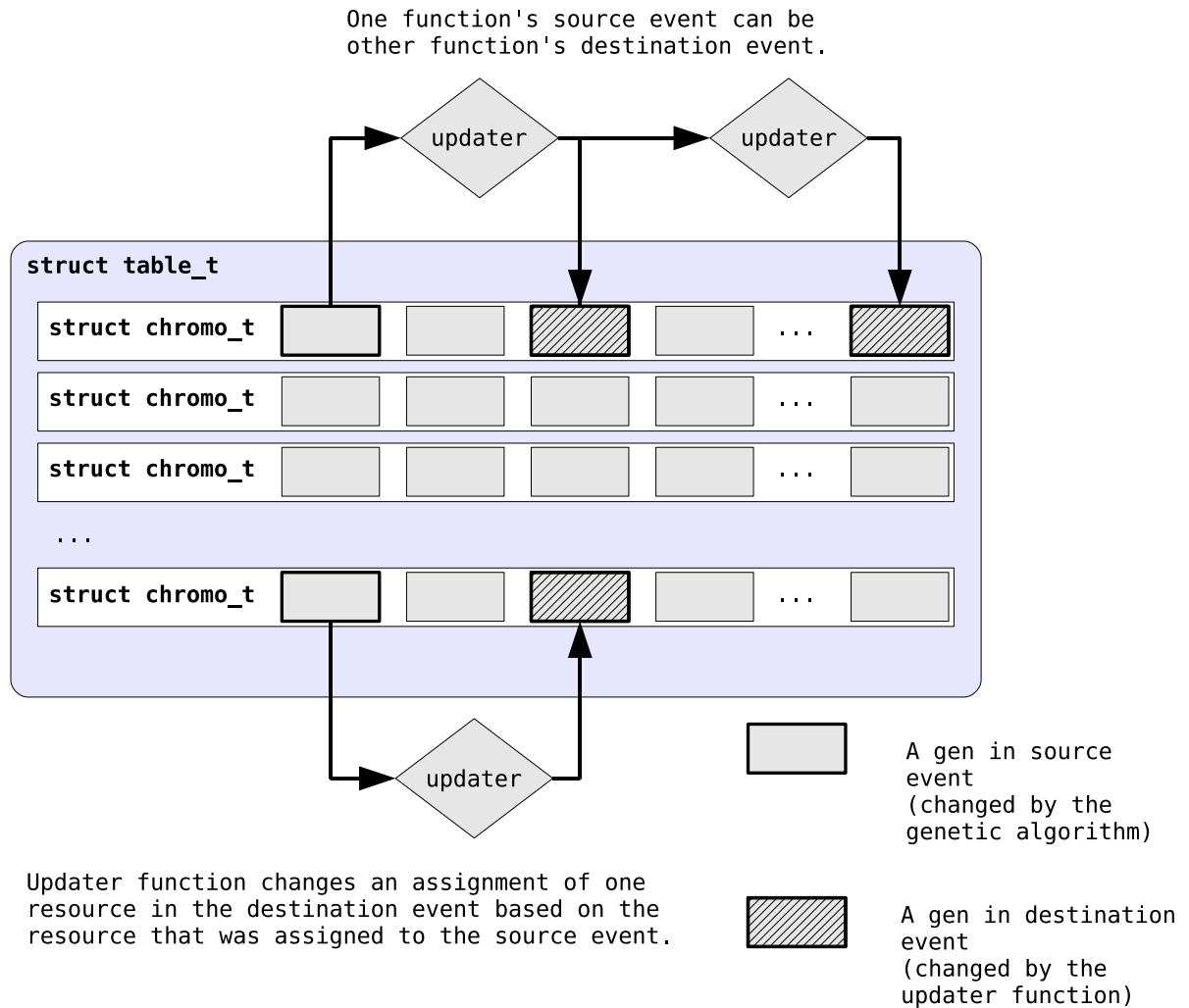
Kernel API for updater functions has changed between releases 0.3.1 and 0.3.2 due to a design error. This makes any modules using the 0.3.1 version of the API incompatible with Tablix kernel 0.3.2 and later. See one of the later sections for more details on porting old modules to the new API).

2.6.2. About updater functions

As you know, the genetic algorithm in the kernel assigns variable resources to events while assignments of constant resources don't change and are set by the problem description file. By defining updater functions a module can take assignment of some or all variable resources for some events away from the kernel and under its own control.

Each updater function basically has a source (independent) event and a destination (dependent). The variable resources of the source event are assigned either by the genetic algorithm or, as we will see later, by another updater function. The updater function then reads these assignments (from the appropriate chromosome structures) and assigns correct variable resources to the destination event.

Figure 2-4. Updater functions in Tablix kernel.



Note: As far as Tablix is concerned, the updater function can use whatever algorithm is needed to calculate the correct variable resources for the destination event, as long as this decision is only based on the assignment of the variable resources to the source event.

Note: A single event can be a source event for more than one function. A single event can also be both a destination event for one function and a source event for one or more functions. However an event can only be a destination event for one updater function.

Note: An updater function must be deterministic. This means that when called with the same inputs it must always return the same output value. In other words, its return value must not depend on any global

variables.

2.6.3. Dependency solving

As you can see a single updater function can only update a single event. Because of this a typical module will define a lot of functions. Also a single problem description can use more than one module that employs updater functions. This means that it is crucial that updater functions are called in the correct order. For example: first functions to be called must be those of which source events are handled by the genetic algorithm. Then those functions can be called that have source events which are also destination events of the first group of functions.

Fortunately you don't need to bother with these details because the Tablix kernel automatically solves any dependency conflicts before starting the main algorithm loop. However you must keep in mind that you should not create any circular dependencies with your functions. Following is an example of a circular dependency:

- Updater function 1: source event 1, destination event 2
- Updater function 2: source event 2, destination event 3
- Updater function 3: source event 3, destination event 1

The dependency solver in the kernel will detect this and report an error.

2.6.4. consecutive.so module

`consecutive.so` module is used to force scheduling of some events in groups of two or more. It requires that a "time" resource type is defined and that it is a matrix.

Since it uses updater functions it contains no fitness functions and therefore also ignores *weight* and *mandatory* parameters.

Following is the important part of the `consecutive.so` source code. Full source code is included in the distribution.

```
/* This structure describes a single consecutive block of events. */
struct cons_t {
    /* Array of tuple ids */
    int *tupleid;
    /* Number of tuple ids in the array */
    int tupleidnum;

    struct cons_t *next;
};

/* Linked list describing all consecutive blocks */
static struct cons_t *cons=NULL;
```

```

static int time;

static int periods, days;

/* This is the updater function. It makes sure that dependent event is
 * scheduled one period later than the independent event. */
int updater(int src, int dst, int typeid, int resid)
{
    return(resid+1);
}

/* This is the precalculate function. It is called after all restriction
 * handlers. We define updater functions here. */
int module_precalc(moduleoption *opt)
{
    int n, tupleid;
    struct cons_t *cur;

    int *residlist;
    int residnum;

    if(cons==NULL) {
        /* The linked list is empty */
        info(_("Module '%s' has been loaded, but not used"),
            "consecutive.so");
    }

    /* We will use this buffer later for the domain_and() function */
    residlist=malloc(sizeof(*residlist)*periods*days);
    if(residlist==NULL) {
        error(_("Can't allocate memory."));
        return(-1);
    }

    /* We walk through all defined groups of event. */
    cur=cons;
    while(cur!=NULL) {

        /* For each event except the first we define an updater
         * function. */

        for(n=1;n<cur->tupleidnum;n++) {
            tupleid=cur->tupleid[n];

            /* We have to check if this event is already dependent.
             * If it is, we report an error. */

            if(updater_check(tupleid, time)) {
                error(_("Event '%s' already depends on another"
                    " event"), dat_tuplemap[tupleid].name);
                free(residlist);
                return(-1);
            }
        }
    }
}

```

```

    }

    /* First event in the group is truly independent
     * (at least as far as this module is concerned). The
     * second event depends on the first. The third event
     * depends on the second and so on. */

    updater_new(cur->tupleid[n-1], tupleid, time, updater);
}

/* Now we have to make sure that the first event in the group
 * will be scheduled so early that the whole group will
 * always fit on the same day. */

/* This means that we have to eliminate the last tupleidnum
 * periods on each day from its time domain. */

residnum=0;
for(n=0;n<days*periods;n++) {
    if(n%periods<(periods-cur->tupleidnum)) {
        residlist[residnum]=n;
        residnum++;
    }
}

tupleid=cur->tupleid[0];
domain_and(dat_tuplemap[tupleid].dom[time],residlist,residnum);

cur=cur->next;
}

free(residlist);
return(0);
}

```

The initialization function, restriction handler and some utility functions are missing from the code listing above. The initialization function merely initializes global variables and registers the restriction handler and precalculate function in the kernel in the usual way. The restriction handler creates a linked list of `cons_t` structures. Each such structure holds a list of tuple IDs (*tupleid* field) of the events that should be consecutive.

2.6.5. Registering updater functions

Updater functions are registered in the Tablix kernel with the `updater_new` function. This function takes four arguments: first is the tuple ID of the source event, second is the tuple ID of the destination event, third is the resource type ID and fourth is the pointer to the actual updater function.

Note: One updater function can only affect assignment of one variable resource (this differs from the 0.3.1 version of the API). The resource type of this variable resource is given as the third argument to `updater_new`.

Before registering an updater function you must check if the destination event is not a destination event of another updater function (from a different module for example). This can be done either with the `updater_check` or by checking the return value of `updater_new`. However the first method is recommended since the `updater_new` can also fail for other reasons (for example because of lack of memory) which can then lead to confusing error messages.

Note: As you can see updater functions are registered in the `precalculate` function. They could also be registered in the restriction handler.

Note: Note that a single physical function is used here in multiple updater functions.

2.6.6. Updater function

The updater function itself is very simple in this case. It is called with three arguments: the tuple ID of the source and destination event and the resource type ID as specified in the `updater_new` function and the resource ID assigned to the source event. It should return the resource ID that should be assigned to the destination event.

Since the module was designed in such a way, the updater function in this case behaves in the same way for all dependent events (it ignores all arguments except *resid*): it assigns a resource to them that has the resource ID of the resource assigned to the source event plus one. Because we are dealing with the time resource type which is a matrix, this means that we are assigning the destination event a timeslot that is just after the timeslot assigned to the source event.

Note: You can't get an assignment of a resource of one resource type to the source event and assign a resource of a different resource type to the destination event.

2.6.7. Updater functions and resource domains

It is possible that when certain resources are assigned to the source event an updater function will try to assign a resource to the destination event that is not in that event's resource domain. Therefore the kernel evaluates each function just before the start of the genetic algorithm and determines for which input values can an updater function be called. Values for which an updater function returns invalid resource assignments are removed from the appropriate resource domains of the source events.

This happens automatically as a part of the dependency solving. In other words the kernel will perform this check for you and you do not need to worry about resource domains when writting updater functions.

2.6.8. Porting modules from 0.3.1 to 0.3.2

In most cases this should be quite straightforward. The updater function in 0.3.2 no longer changes the table structure directly. It gets the assignment of the requested resource to the source event as an argument and returns the resource ID of the resource that should be assigned to the destination event.

Because of this change, a single updater function can only affect assignments of resources of one resource type. `updater_check` and `updater_new` now have one more argument to deal with this change.

Following are the relevant parts of the `consecutive.so` module before the change (compare them to the listing above):

```
void updater(int src, int dst, table *tab)
{
    int src_time;

    src_time=tab->chr[time].gen[src];

    /* This should always be true if we correctly set the time domain
     * for the first event in each group. */

    /* It means that the independent event is not on the last period
     * of the day. */

    assert(src_time%periods<(periods-1))

    /* Next event in the group is scheduled one period later. */

    tab->chr[time].gen[dst]=src_time+1;
}

.
.
.

int module_precalc(moduleoption *opt)
{
    .
    .
    .

    /* We have to check if this event is already dependent.
     * If it is, we report an error. */

    if(updater_check(tupleid)) {
```

```

        error(_("Event '%s' already depends on another"
               " event"), dat_tuplemap[tupleid].name);
        free(residlist);
        return(-1);
    }

    /* First event in the group is truly independent
     * (at least as far as this module is concerned). The
     * second event depends on the first. The third event
     * depends on the second and so on. */

    updater_new(cur->tupleid[n-1], tupleid, updater);

    .
    .
    .
}

```

2.7. Miscellaneous

2.7.1. Order of calling

Various functions defined by fitness modules are called in the following order:

1. After kernel initialization the XML configuration file is parsed by the `parser_main()` function. All defined resource types are stored in the `dat_restypes` array. All defined resources are stored in resource lists under appropriate `resourcetype` structures. All defined events are stored in `dat_tuplemap` array. `dat_info` is filled with values from the `<info>` tag.
2. `<modules>` tag is parsed. Modules are loaded into memory and `module_init()` function is called in each module. By this time all arrays and structures mentioned above have been initialized and will not change anymore.
3. `<restriction>` tags for resources are parsed. For each tag all corresponding handler functions are called.
4. `<restriction>` tags for events are parsed. For each tag all corresponding handler functions are called.
5. All registered precalculate functions are called.
6. The genetic algorithm starts. Any registered fitness functions may now be called by the algorithm with appropriate arguments.

Chapter 3. Module documentation

3.1. Introduction

Tablix distribution includes a specialized automatic documentation system similar to Doxygen. `mod-doc2.pl`, a Perl script in the `doc` directory, parses the source code of Tablix fitness modules and produces a module reference manual in HTML form. This script is smart enough to deduce the following information directly from the source code:

- Registered resource and tuple restrictions,
- supported module options,
- resource types on which the module depends.

Information that cannot be gathered from the source code can be provided to the documentation system in a form of one or more specially formatted comment blocks. These blocks can include information on:

- Module author,
- description of functionality provided by this module,
- description of registered restrictions and supported options,
- list of module groups this module belongs to.

Note: It is not mandatory to include this information in the module source code. The documentation system will always make a page in the reference manual for a module even if it doesn't contain any special comment blocks. However all modules that are to be included in the official Tablix distribution must include this information (documenting your module is a good idea in any case).

3.2. Comment block syntax

Following is an example comment block. Each module source file can include one or more such comment blocks. Comment blocks can appear in any part of the source code.

```
/** @block-tag This text is associated with the block-tag tag.
 *
 * @keyword-tag There can be any number of keyword-tags in a comment block.
 *
 * @keyword-tag Text can appear in the same line as the keyword-tag...
 * Or in any line bellow it.
 *
 * Paragraphs are separated with a single blank line.
 */
```

Each comment block must start with `/**` (note two asterisks) and end with `*/`. It can span multiple lines. Leading whitespace with an optional asterisk is ignored on lines within the comment block.

Each comment block can include one or more tags. Tags are words prefixed with a `@` character. The first tag in a comment block is called block tag and marks the role of the current comment block. Any subsequent tags are called keyword tags and are used to delimit various parts of the comment block. Which keyword tags can appear in a comment block depends on the type of block (i.e. the block tag of this comment block).

Text is associated with a block or keyword tag that appears in front of it. Text can start on the same line as the tag or in the following lines. Some tags can accept multiple paragraphs of text (for example the `@brief` tag). In that case paragraphs are separated with blank lines.

3.3. Module information block

Following is an example module information block from the `preferred.c` module source code. Each module can include at most one module information block.

```
/** @module
 *
 * @author Tomaz Solc
 * @author-email tomaz.solc@tablix.org
 *
 * @credits
 * Ideas taken from a patch for Tablix 0.0.3 by
 * Jaume Obrador <obrador@espaiweb.net>
 *
 * Ported to version 0.2.0 and extended by Nick Robinson <npr@bottlehall.co.uk>
 *
 * @brief Adds a weight whenever an event is not scheduled at the specified
 * preferred day and/or time slot.
 *
 * @ingroup General
 */
```

Module information block uses `@module` block tag. This block tag does not take any arguments, so no text needs to be entered between the block tag and the first keyword tag. Following keyword tags are supported in this block (No tags are mandatory. At most one tag of each type per comment block, unless stated otherwise):

`@author`
Name of the author or maintainer of this module.

`@author-email`
Email of the author or maintainer.

@credits

In case of multiple authors you can enter acknowledgments in this keyword tag. Supports multiple paragraphs of text.

@brief

Description of the module's functionality. Supports multiple paragraphs of text.

@ingroup

Defines groups of modules this module belongs to. Group names must be separated by commas.

3.4. Restriction and module option information blocks

Documentation for restrictions and module options can be entered in one or more comment blocks. Following is an example block documenting the *preferred-period* tuple restriction from the `preferred.c` module source code.

```
/** @tuple-restriction preferred-period
 *
 * <restriction type="preferred-period">period</restriction>
 *
 * This restriction specifies the preferred period for an event.
 */
```

Following block tags can be used to document restrictions and options:

@tuple-restriction

Used for tuple (event) restrictions. Must be followed by the name of the restriction, as defined by the `handler_tup_new()` function.

@resource-restriction

Used for resource restrictions. Must be followed by the name of the restriction, as defined by the `handler_res_new()` function.

@option

Used for module options. Must be followed by the name of the option, as used in `option_find()`, `option_int()` or `option_str()` functions.

Restriction and module option information blocks do not accept any keyword tags. The entire content of the comment block is considered a description of the restriction or module option.

Multiple paragraphs of text are supported in these comment blocks. XML configuration examples can be inserted in the text without any special keyword tags. Any paragraph where all lines begin with valid XML tags is considered an example and is printed in monospace font in the reference manual.

Note: One restriction or module option is documented per comment block. The block tag differs depending on the type of the restriction.

Chapter 4. Tablix Testing Framework

4.1. Introduction

Tablix testing framework provides a way to write simple automated tests that verify if a module and/or kernel is working as expected. Framework is composed of a special export module `export_ttf.so` and a utility program `tablix2_test`. You can find both in the `ttf/` subdirectory of the Tablix source tree.

Each automated test case usually verifies a single function of a module and is stored in a single file with the standard Tablix XML configuration syntax and a special XML comment block. This block contains a short program written in Scheme. The program is then used to verify the timetable that is produced by Tablix.

Note: Tests for all modules that are included in the Tablix distribution are stored in the `ttf/tests/` subdirectory. Most modules are complicated enough that they require more than one test case. Files with tests are named `modulename-1.xml`, `modulename-2.xml`, `modulename-3.xml`, etc. (where `modulename` is the name of the module to test).

Note: Scheme code in the test case has a similar role to the fitness function in the module code. The main difference is that the fitness function returns an integer (higher value means more errors) and the scheme code returns only true (test passed) or false (test failed).

You may ask why an obscure language like Scheme was used for this purpose. There are two reasons for this decision: First, TinyScheme (<http://tinyscheme.sourceforge.net/>) interpreter is only 4500 lines long, is very easy to embed and extend, is safe if used as prescribed and is available under a BSD license. Second, the purpose of a test would be defeated if a programmer could copy and paste code from the module to the test. A programming language with a completely different syntax than C forces the author to reimplement the algorithm in a different way. This minimizes the chances that two implementations (the actual module and the test) would have the same errors.

4.2. How a test is processed

This section describes what happens when you run the `tablix2_test` utility.

First, the test file is processed by Tablix as if it was a normal XML configuration file. Tablix finds a solution to the problem that is described in it and saves it in a result file.

The result file is then parsed again. The constructed timetable is read from the XML result file into internal data structures. The Scheme code segment is found and interpreted. The scheme program checks the timetable data for errors returns either true or false (depending whether the test was passed or failed).

4.3. A basic test

Let's write a simple "Hello, world!" test. Open a file named `hello-1.xml` and enter the following lines:

```
<?xml version="1.0" encoding="iso-8859-1"?>

<!-- BEGIN TTF BLOCK
(display "debug: Hello world!")
(newline)
(test-ttf
  #f
)
END TTF BLOCK -->

<ttm version="0.2.0">
  <modules>
    <module name="hello.so" weight="10" mandatory="yes"/>
  </modules>

  <resources>
    <variable>
      <resourcetype type="dummy-type">
        <linear name="#" from="1" to="50"/>
      </resourcetype>
    </variable>
  </resources>

  <events>
    <event name="dummy-event" repeats="1">
    </event>
  </events>
</ttm>
```

You can see that most of this file is a normal Tablix XML configuration (using our "Hello world!" fitness module from the second chapter). If you would process this file with Tablix it would work without problems because the Scheme code is commented out and the XML parser will ignore it. You can also see that the scheme code segment is delimited with lines containing words `BEGIN TTF BLOCK` and `END TTF BLOCK`.

Tip: If you are not familiar with scheme, I recommend reading the Guile Reference Manual (<http://www.gnu.org/software/guile/docs/guile-ref/index.html>). Beware that TinyScheme does not implement everything that is described there.

For the impatient: The general syntax is like the following

```
(function argument1 argument2 ...)
```

Functions are defined like this

```
(define (function arg1 arg2 ...) code ...)
```


A typical "for" loop where *i* goes from 1 to 14 consists of a recursive function that calls itself 15 times (recursion is used a lot in Scheme):

```
(define (loop i)
  (if (< i 14)
      (begin
        code ...
        (loop (+ i 1))
      )
      )
  )
(loop 1)
```

Use a semicolon to make comments in the code, like this:

```
; This is a comment
```

If you look at the code you can see that we calling three functions in our test. First two functions are here for demonstration purposes only and aren't usually used in tests (unless you are debugging your Scheme code, in which case you will use them a lot): `(display)` function prints some text to the standard output. `(newline)` prints a newline character after the text.

`(test-ttf)` is a function which must be called in every test. It can have an arbitrary number of arguments and all of them must evaluate to `#t` (boolean "true" in Scheme) if the module passes this test. Since we pass `#f` (false) in this example, the test will obviously fail each time.

You can now run this test using the following command line:

Note: See the `tablix2_test(1)` man page for detailed description of `tablix2_test` command line arguments. For the moment only note that the second argument always contains a list of options to be passed to Tablix. If you don't need any options (which is true in most cases) then this argument must still be present, but empty. You can specify an empty argument in `bash` shell with `" "`.

```
$ tablix2_test --file "" hello-1.xml
```

```
TABLIX testing framework 0.2.1, Copyright (C) 2002-2005 Tomaz Solc
```

```
using binary : /home/avian/software/bin/tablix2
using binary : /home/avian/software/bin/tablix2_output
repeats      : 1
parameters   :
test file     : hello-1.xml
```

```
hello-1 : test-ttf: test number 1 failed
debug: Hello world!
*** FAILED (ttf test failed) ***
```

You can see that the test failed as expected, but the greeting was printed anyway. To keep the noise down, only lines that begin with "debug" are actually printed by the `tablix2_test` utility, and even these are only printed if the test failed.

4.4. How to write scheme code

As mentioned above, each test must include exactly one call to function `(test-ttf)`. Arguments to this function must all evaluate to true (or `#t` in scheme syntax) if the test is successful. If one or more arguments are false, the `tablix2_test` utility will say that the test was not successful and print out which argument was not true.

This means that each test can be composed of several sub tests. Each sub test can be in its own function that returns either `#t` if the test was successful or `#f` if it was not. The functions are then used as arguments to `test-ttf`.

For each resource type defined in the Tablix XML configuration two Scheme functions are defined by default:

`(get-resourcetype tuple-id)`

This function returns an integer, equal to the resource ID of the resource of the resource type *resourcetype* that is used by event with tuple ID *tuple-id*.

Integer *tuple-id* can be replaced with a string holding a valid event name (as used in the *name* property in the XML configuration). If more than one event with that name is found (for example if *repeats* is greater than 1), resource ID of the resource used by the first matching event is returned.

`(resourcetype tuple-id resource-id-1 [resource-id-2])`

The third argument is optional. If this function is called with two arguments, then it returns `#t` (boolean true) if event with tuple ID *tuple-id* is using a resource with resource ID *resource-id-1* and resource type *resourcetype*. If the third argument is present then it return `#t` if this event is using any resource with resource ID between *resource-id-1* and *resource-id-2*.

Again, integer *tuple-id* can be replaced with a string holding a valid event name. If more than one matching event is found, only the first is checked. Integer arguments *resource-id-1* and *resource-id-2* can also be replaced with strings holding valid resource names.

You can use these two functions either directly in arguments to the `test-ttf` function or in your own defined functions to check the result timetable produced by Tablix.

4.5. A test case for the `fixed.so` module

Let's write a TTF test case for the `fixed.c` module we wrote in chapter 2. First we have to write a valid XML configuration file that uses this module. In this case we can use our example configuration file `fixed.xml`. Copy it to a new file `fixed-1.xml`.

Now we have to add the scheme code in the comment block. Since our module is very simple (it only forces some events to use a specific variable resource) we can check if the solution is correct just by calling the `dummy-type` function. In this case we have to check only if the first defined event is using the resource with the name "30".

```
<?xml version="1.0" encoding="iso-8859-1"?>
```

```

<!-- BEGIN TTF BLOCK
(test-ttf
  (dummy-type 0 29)
)
END TTF BLOCK -->

<ttm version="0.2.0">
  <modules>
    <module name="fixed.so" weight="10" mandatory="yes"/>
  </modules>

  <resources>
    <variable>
      <resourcetype type="dummy-type">
        <linear name="#" from="1" to="50"/>
      </resourcetype>
    </variable>
  </resources>

  <events>
    <event name="dummy-event" repeats="1">
      <restriction type="fixed-dummy-type">30</restriction>
    </event>
  </events>
</ttm>

```

The first argument to the `dummy-type` is the tuple ID of the only event defined in this configuration file. The second argument is the resource ID that this event should be using. Since resource IDs are numbered from 0 and we named the resources with numbers starting from 1, the resource ID is equal to the name of the resource minus 1.

Tip: If you are unsure how the tuple IDs or resource IDs are numbered first pass your configuration file to Tablix and have a look at the result (`result*.xml`) file. All `<resource>` and `<event>` tags in this file will have `res-id` and `tupleid` properties assigned to them showing how Tablix allocated IDs to events and resources.

Passing tuple IDs and resource IDs to functions can be useful if more than one tuple or resource is checked in a loop. Because we are only checking a single tuple for a single resource, we can make the Scheme code a bit more readable by replacing numerical IDs with tuple and resource names. The modified code would look like this:

```

<!-- BEGIN TTF BLOCK
(test-ttf
  (dummy-type "dummy-event" "30")
)
END TTF BLOCK -->

```

4.6. A test case for the `consecutive.so` module

This is an example if a module that requires a bit more Scheme code to test (the `consecutive.so` module forces Tablix to schedule some events in consecutive time slots on the same day - see the Module reference manual for details)

Following are the contents of the test case `consecutive-1.xml` (you can find it in the `ttf/tests/` subdirectory).

```
<!--
BEGIN TTF BLOCK
(define (tuple-loop i j ok)
  (if (< i j)
      (tuple-loop (+ i 1) j
                  (and
                   ok
                   (time i (+ (get-time (- i 1)) 1))
                   (=
                    (quotient (get-time i) 10)
                    (quotient (get-time (- i 1)) 10)
                   )
                  )
      )
      ok
  )
)
(test-ttf
 (tuple-loop 2 6 #t)
)
END TTF BLOCK
-->
<ttm version="0.2.0">
  <modules>
    <module name="sametime.so" weight="60" mandatory="yes"/>
    <module name="timeplace.so" weight="60" mandatory="yes"/>
    <module name="consecutive.so" weight="60" mandatory="yes"/>
  </modules>

  <resources>
    <constant>
      <resourcetype type="teacher">
        <resource name="a"/>
      </resourcetype>
      <resourcetype type="class">
        <linear name="#" from="1" to="3"/>
      </resourcetype>
    </constant>
    <variable>
      <resourcetype type="room">
        <linear name="#" from="1" to="40"/>
      </resourcetype>
    </variable>
  </resources>
</ttm>
```

```

        <resourcetype type="time">
            <matrix width="10" height="10"/>
        </resourcetype>
    </variable>
</resources>

<events>
    <event name="test" repeats="1">
        <resource type="teacher" name="a"/>
        <resource type="class" name="1"/>
    </event>
    <event name="test" repeats="5">
        <resource type="teacher" name="a"/>
        <resource type="class" name="2"/>
        <restriction type="consecutive"/>
    </event>
    <event name="test" repeats="1">
        <resource type="teacher" name="a"/>
        <resource type="class" name="3"/>
    </event>
</events>
</ttm>

```

`consecutive.so` module uses a timetable extension in its fitness function. Because of that we have to use the `sametime.so` and `timeplace.so` modules in the test case or the `consecutive.so` module will not work correctly.

If you take a look at the XML configuration you will see that the middle five events must be scheduled in consecutive time slots. These events have tuple IDs from 1 to 5.

Scheme code consists of a recursive loop function `tuple-loop`. *i* holds the tuple ID we are currently checking, *j* holds the last tuple ID to check + 1 and *ok* holds the final return value (#t if the tuples so far have been consecutive or #f if not).

If we look at the `tuple-loop` in detail we can see that first we have an if conditional to stop the recursion once we check all tuple IDs. If that is true, we return the return value in *ok*. If not, we check the current tuple *i* and call `tuple-loop` with incremented tuple ID *i* and an updated return value *ok*.

The new return value *ok* is #t if all the previous tuples were consecutive (old *ok* is #t), if the current tuple is scheduled one time slot later than the previous tuple and the current and previous tuples are scheduled on the same day (see section on matrices).