

Test development with T2C: getting started

Index

- I. Overview
- II. Typical Workflow
- III. Case Study
 - Prerequisites
 - Analysis of the documentation and interface grouping
 - Requirement markup
 - Generating T2C file template
 - Populating the T2C file template
 - Creating requirement catalogue
 - Generating C code and building the tests
 - Executing and debugging the tests
- IV. Notes
- V. Glossary

I. Overview

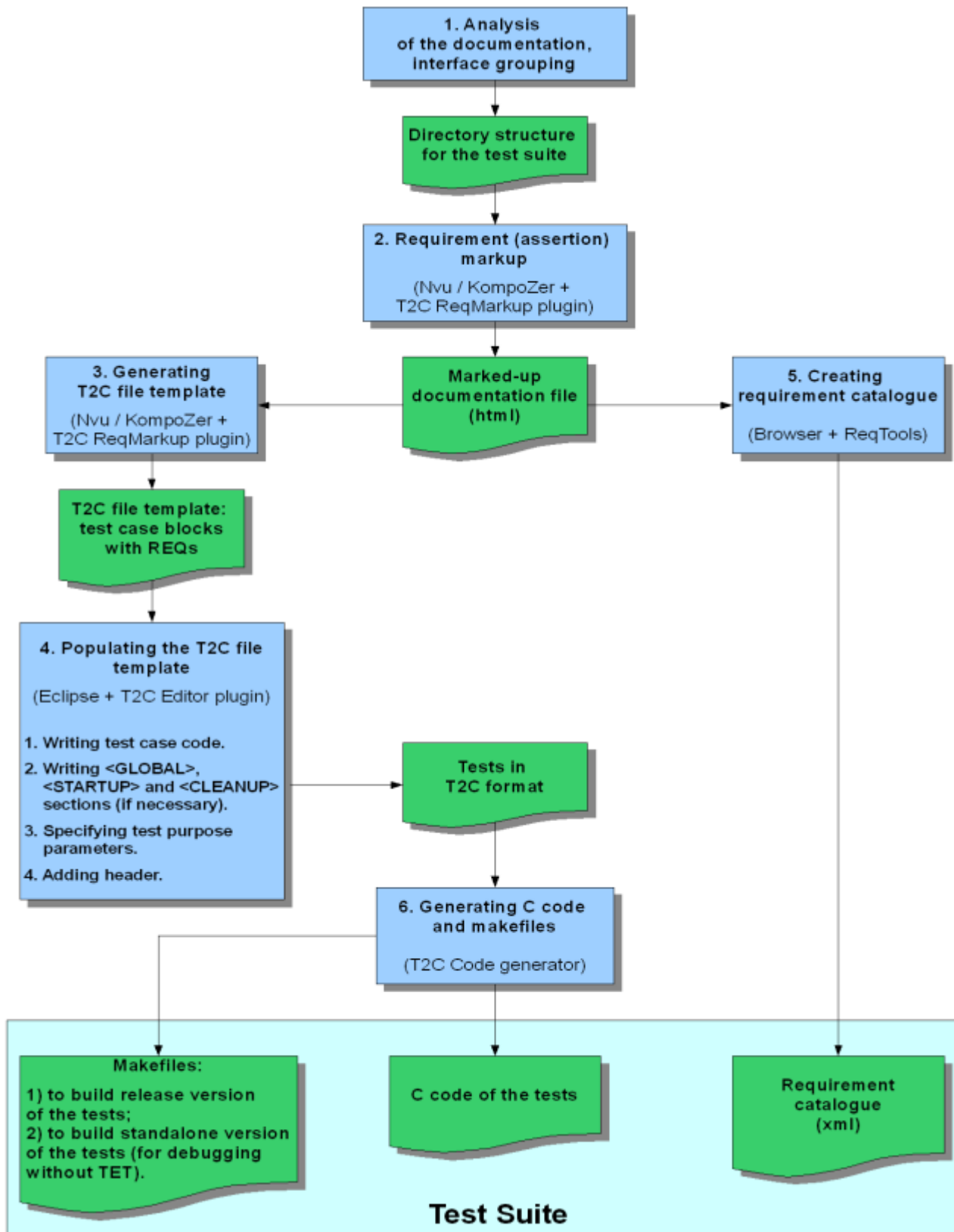
The T2C ("Template-to-C") framework is designed for development of the "normal"-quality tests (see part V) that can be used with the TETware Lite test harness (http://tetworks.opengroup.org/Products/tetware_lite.htm).

Key features of T2C

- Automatic generation of test purpose functions for each parameter set based on a test template ("a document with holes"). The generated C code is structured. It is clear, which interfaces are tested in a particular test purpose and which values of the parameters are used in this test purpose.
- Requirement checks are linked to the specific places in the documentation. Requirement text (perhaps reformulated to improve readability) is output to the TET journal in case of failure. So during analysis of the TET journal it is easier to figure out what happened. Coverage analysis is also easier. (Each requirement has a unique ID which is output to the journal when the requirement is checked.)
- Automatic generation of common TET support code: defining TET structures (e.g. `tet_testlist`) in a consistent way, etc.
- Requirement (assertion) checks via special macros and functions, more convenient than using bare TET API.
- Each test (test purpose) is executed in a separate process and thus it generally has far less effect on the other tests.
- Maximum execution time can be specified. This can be useful if some tests may hang. (If they do, at least other tests will have a chance to be executed.)
- Both global and per-test resource cleanup support.
- Support for building a test executable that does not depend on TET features at all (no change in the code of the test required). This can be useful, for example, to debug the test or to obtain more data about the behaviour of the tested interfaces in case of failure.

II. Typical Workflow

The T2C ("Template-to-C") framework is designed for development of the "normal"-quality tests (see part V) that can be used with the TETware Lite test harness (http://tetworks.opengroup.org/Products/tetware_lite.htm). Test development with T2C usually consists of the following steps.

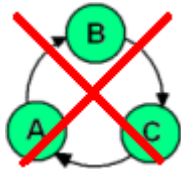


1. Analysis of the documentation and interface grouping

First of all, before trying to write conformance tests for some interfaces one should examine the documentation of these interfaces to find out what is to be tested.

Then the interfaces to be tested should be split into some functional groups. Sometimes the grouping is already done in the documentation. For instance, “Glib Reference Manual” (<http://www.gtk.org/api/2.6/glib/>) is divided in sections such as "Memory Allocation", "String Utility Functions", "Key-value file parser", etc. Interfaces described in each section form a single functional group in this case.

The functional groups of interfaces should not be very large. The recommended maximum size of a group is 10–15 interfaces.



Ideally, the functional groups should have no cyclic dependencies between each other. Group "A" depends on group "B" if we need interfaces from "B" to test those from "A". If "A" depends on "B", "B" depends on "C" while "C" depends on "A" (or even "B" depends on "A") it is a cyclic dependency. Situations like this should be avoided.

For each group of interfaces a T2C file will be created and placed in a proper place in the special directory structure (see III.1). The T2C file will contain the tests for these interfaces in a format described below (see II.3). The appropriate directory structure for the test suite is also created at this stage.

The documentation for the interfaces is assumed to be in html format.

2. Requirement (assertion) markup

At this stage the so called elementary requirements or elementary assertions (see part V) are extracted from the documentation. Each requirement is given a unique ID and is linked to an excerpt from the documentation. The excerpt is marked up in a special way in KompoZer (or Nvu) html editor enhanced with T2C ReqMarkup plugin.

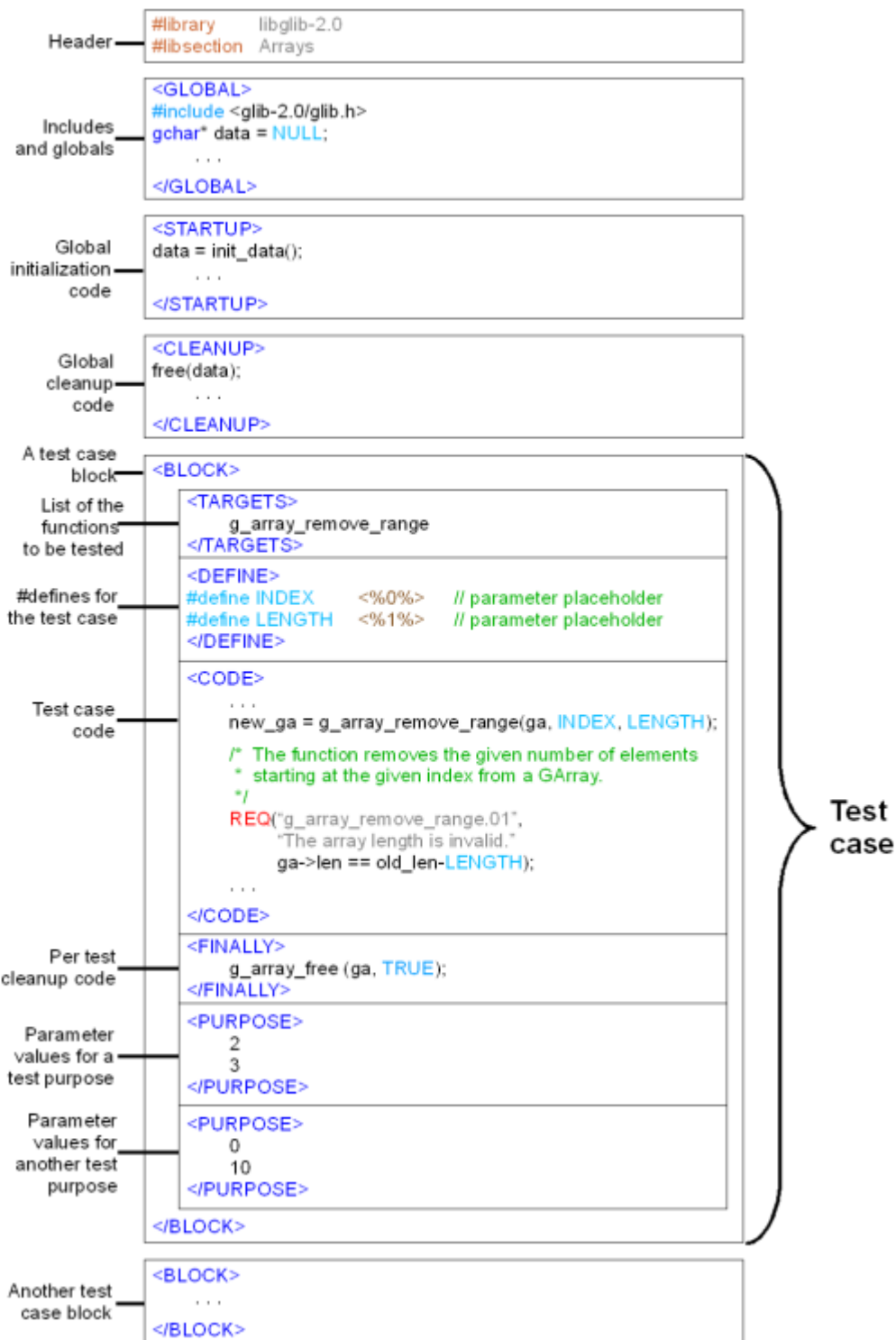
By default this excerpt represents the requirement's text associated with the respective ID. The text can be reformulated if necessary to improve readability. This reformulated text does not replace the original one in the documentation, it affects only requirement's textual representation associated with a given ID. The reformulated requirement's text (or the original text if the reformulated text is not specified) is output to the TET journal in case of an assertion failure.

The result of this step is the html file with the documentation enriched with the marked up requirements for the interfaces.

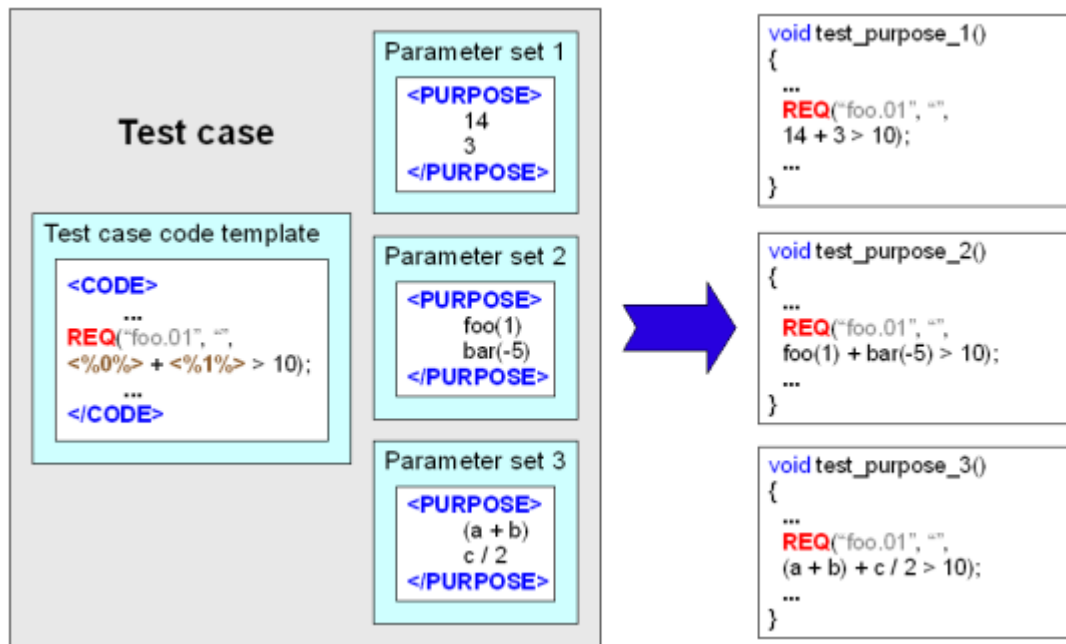
3. Generating T2C file template

Once the requirements have been marked up, the ReqMarkup plugin generates a template for the test in the T2C format.

Overall structure of a T2C file is shown below:



The sections of a T2C file are marked up with special tags: `<GLOBAL> ... </GLOBAL>`, `<CODE> ... </CODE>`, etc. The code in these sections is a plain C, although the test case code (`<CODE>` section) can also contain placeholders for the test purpose parameters. The sets of such parameters are specified in the `<PURPOSE>` sections. During the step 6 the T2C code generator will create a C function for each test purpose by substituting the actual parameter values instead of these placeholders. That is, the contents of the `<CODE>` section will be used as a template for the C code of the test.



For each interface a test case template is created with a call of REQ macro for each requirement (see III.4.6.1). REQ is used to check the requirement and report failures. It receives a requirement ID, an expression to evaluate and some other data as its arguments. If the expression evaluates to 0, it means that the requirement (assertion) has failed. In this case the appropriate message is output to the TET journal along with the requirement text and the test result code is set to FAIL.

4. Populating the T2C file template

This is the most important phase when the tests in the T2C format are written.

If it is necessary to provide some global data for the tests, perhaps along with the code for its initialization and cleanup, it can be specified in the <GLOBAL>, <STARTUP> and <CLEANUP> sections of the T2C file, respectively.

The T2C file header specifying the name of the library and the functional group of interfaces under test should also be provided.

The T2C Editor Eclipse plugin can be helpful for visual editing of T2C files providing advanced navigation among the sections along with other useful features. However, T2C files can be edited with any text editor as well. The development of the T2C Editor plugin is now in progress.

It is not recommended to use TETware Lite API directly from the test case templates. Macros and API functions provided by the T2C support library are higher level means to perform common operations.

5. Creating requirement catalogue

Based on the html with the requirements marked up (see II.2), a requirement catalogue is created. This catalogue is used by the tests in failure reporting. It contains the ID and the text (original or reformulated, if specified) for each requirement.

The catalogue is created by the ReqTools javascript (see III.2.1) in a web browser. (For the present, it has been tested for Mozilla Firefox, Opera and Internet Explorer. It may not work properly in Konqueror though.)

The currently used catalogue file format is rather simple. The catalogue file is an xml file that contains an ordinary header ("<?xml version="1.0"?>"), a root element ("requirements") and one or more "req" elements within the root element. Requirement id is specified in the "id" attribute of this tag, while its body is the requirement's text (see II.2) Consider the example below:

```
<?xml version="1.0"?>
<requirements>
<req id="atk_streamable_content_get_mime_type.01">
```

```

Returns a gchar* representing the specified mime type.
</req>
<req id="app.atk_streamable_content_get_mime_type.02">
streamable: a GObject instance that implements AtkStreamableContentIface.
</req>
<req id="app.atk_streamable_content_get_mime_type.03">
i: a gint representing the position of the mime type starting from 0.
</req>
<req id="atk_streamable_content_get_n_mime_types.01">
Returns a gint which is the number of mime types supported by the object.
</req>
<req id="app.atk_streamable_content_get_n_mime_types.02">
streamable: a GObject instance that implements AtkStreamableContentIface.
</req>
<req id="atk_streamable_content_get_stream.01">
Returns a GIOChannel that contains the content of the specified mime type.
</req>
<req id="app.atk_streamable_content_get_stream.02">
streamable: a GObject instance that implements AtkStreamableContentIface.
</req>
<req id="app.atk_streamable_content_get_stream.03">
mime_type: a gchar* representing the mime type.
</req>
</requirements>

```

Sometimes the requirements for a group of interfaces are specified in several html files rather than one. In this case the complete requirement catalogue for this group should be assembled manually from the parts (xml files) generated by the ReqTools script for each of the html files with the documentation. (That is, these xml files should be merged into a single xml file.)

6. Generating C code and makefiles

Now that the tests in T2C format and the requirement catalogue files are prepared, it is time to invoke the T2C code generator to finally create the C code of the tests as well as necessary makefiles for individual tests and the test suite as a whole.

For this to work properly, T2C files, requirement catalogue files and other files should be placed in a proper directory structure (described in III.1 and IV.C1).

Once the directory structure is prepared and each file is where it should be, one can run the code generator.

The code generator also creates special targets in the generated makefiles that can be used to build a test without using the TET test harness (see the details in III.7.2). The test executable built in this way does not depend on TET at all which can simplify debugging. There is no free lunch, of course. Many of the TET features are not available in this case such as advanced signal handling and error reporting, etc.

7. Building, executing and debugging the tests

The test suite is ready for building. One can now build it using the makefiles generated at the previous step, then run the tests or debug some of these outside the TETWare Lite test harness.

III. Case Study

Let's consider the following examples. Assume we want to develop tests for the two group of interfaces:

1. Glib Arrays (Glib Reference Manual, "Arrays" - <http://www.gtk.org/api/2.6/glib/glib-Arrays.html>)
2. AtkStreamableContent (ATK Library, "AtkStreamableContent" - <http://www.gtk.org/api/2.6/atk/AtkStreamableContent.html>)

Test development for the first group is described in detail below. As for the second one, its only purpose is to demonstrate some important features that are not used in the 1st example (such as common include files, access to test data, etc.). These features are described later.

Prerequisites

For this case study you need the following. To markup the requirements and create requirement catalogue:

- KompoZer HTML editor (<http://www.kompozer.net/>) v.0.7.7 or higher See also important notes in IV.M1.
- T2C ReqMarkup plugin for KompoZer/Nvu (can be found in **samples/tools/reqmarkup/**). Installations instructions are available IV.M2.
- T2C ReqTools script (**samples/tools/reqtools/reqtools.js**)

To build and execute the tests:

- LSB 3.1 Software Development Kit (<http://www.linux-foundation.org/en/Downloads>).
- LSB-compliant version of TETware Lite test harness: lsb-tet3-lite and lsb-tet3-lite-devel packages (see the latest shapshots here: <ftp://ftp.linux-foundation.org/pub/lsb/snapshots/tet-harness/>).
- GNU C/C++ Compiler 3.4.6 or newer.
- pkg-config
- libtool
- Glib2 library
- ATK library

The source code of the examples can be found in the **samples** subdirectory along with necessary tools described below.

1. Analysis of the documentation and interface grouping

All the 12 glib array interfaces that are present in the LSB 3.1 form a single functional group (see II.1).

Before we create a T2C file for it, consider the directory structure of the sample test suite. (The general test suite structure is described in IV.C1) The root of this directory structure is assumed to be specified in the T2C_ROOT environment variable.

The **samples** directory contains subdirs for each sample subsystem: **sample-01-t2c** for the 1st example and **sample-02-t2c** for the 2nd one. There are also **tet_code** and **tetexec.cfg** files in this directory that are required to execute the tests under TET.

There are several scripts here too:

- **build_all.sh** - executes the T2C C code generator and then builds the newly created C sources using make.
- **run_tests.sh** - executes the tests using the test case controller (tcc) provided by TET.
- **clean_all.sh** - removes all files and directories **build_all.sh** has created.

It is not mandatory to use these script but they can be useful.

build_all.sh and **clean_all.sh** use the **gen_tests** and **clean** scripts found in the respective subsystem directories.

A directory for each subsystem (**sample-01-t2c** and **sample-02-t2c**) should contain the following files and subdirectories:

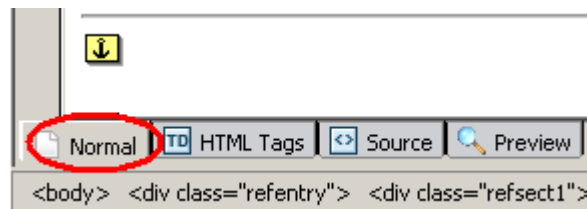
- **a configuration file (*.cfg)** for the T2C code generator – see IV.C3.
- **gen_tests** - test generation script.
- **clean** - clean up script.

- **src/** - sources of the tests (in T2C format) for the respective library. Each subdirectory contains tests for a respective group of interfaces.
- **include/** (if present) - a common directory containing header files for the tests (see IV.C5 for an example).
- **reqs/** - requirement catalogue.
- **testdata/** (if present) - data used for testing (along with the sources and makefiles if the data needs to be built, e.g. special modules, etc.). The subdirs of this directory have the same names as the t2c-files with tests, which data they contain. (For instance, **samples/sample-02-t2c/testdata/AtkStreamableContent** contains data for the tests from **samples/sample-02-t2c/src/atk_samples/AtkStreamableContent.t2c**.) The path to the test data can be obtained in the test code via T2C_GET_DATA_PATH() macro (see IV.C4).
- **scenarios/** (created by the T2C C code generator) - TET scenario files.
- **tests/** (created by the T2C C code generator) - generated C files with the tests, makefiles necessary to build them.

2. Requirement markup

Let us markup elementary requirements (see part V) for the behaviour of g_array interfaces in the documentation. We will use KompoZer enhanced with T2C ReqMarkup plugin.

Open **samples/html/sample-01/glib-Arrays.html** in KompoZer. Switch to Normal view if it is not already active.



For now you need not worry about how these requirements are going to be tested. Leave this for step 4 and just mark them up as described below.

Only those parts of the text that contain the requirements for the behaviour of the interfaces are marked up. That is why the following fragments should NOT be marked up in this case:

- general description of the interfaces dealing with glib arrays ("Description"): there are no particular requirements for the interfaces in it, the requirements are stated below in the descriptions of these interfaces;
- the example "Example3".

The results of the requirement markup for the "Glib Arrays" interface group are in this file: **samples/html/sample-01/results/glib-Arrays.html**

2.1. g_array_new

Let us begin with g_array_new.

- Select a continuous text block containing the requirement: *"Creates a new GArray."*
- Press Ctrl+R, Alt+R or select "Assign REQ" in the context menu, on the toolbar or in the "Insert" menu.

•

The image shows a 'REQ Properties' dialog box. It has a title bar with a close button. Inside, there's a 'Requirement ID(s)' field containing 'g_array_new.01' and a '+' button. Below this is a list of 'Existing IDs' (empty) and a 'Filter' field. To the right, there's a 'Requirement text' field containing 'Creates a new GArray.' and an 'Alternative text' field (empty). At the bottom, there are 'Delete', 'Add', and 'Cancel' buttons.

- Specify requirement identifier. Each requirement we mark up should have a unique identifier. It will be used in the requirement catalogues (see II.5), output to the TET journal in case of failure, etc. Most of the time it has the following syntax: *<function>.NN*, where NN is a number (an "index") containing at least 2 digits, e.g.: *g_array_new.01*. The numbers should start from 01 although it is not mandatory. More complex cases will be explained later as we encounter them.

So the requirement contained in the selected fragment now has an id: *g_array_new.01*.

- The "requirement's text" field shows the original text of the requirement (actually, the text we have just selected). We can reformulate it to make it clearer if necessary and associate the reformulated text with this requirement. Use the "Alternative text" field for that. The meaning of the requirement should remain the same, of course.

It is the requirement's text (original or reformulated - if specified) that is output to the TET journal in case of a failure.

- Press "Add". Now the text block we selected represents a requirement with ID *g_array_new.01*.

The text blocks we have marked up are highlighted in yellow. To toggle the highlighting, press "Highlight REQ" on the toolbar.

There is another requirement for *g_array_new* in the description of 'zero_terminated' parameter: *"TRUE if the array should have an extra element at the end which is set to 0."*. The requirement can be reformulated this way: *"If 'zero_terminated' is TRUE, the array shall have have an extra element at the end which is filled with 0s."*

Like *g_array_new.01*, select the appropriate text block in the documentation, press Ctrl+R, then edit the ID (you may leave the id suggested by ReqMarkup: *g_array_new.02*).

The "+" button increments by 1 the last index in the ID. For example, "foo.02" becomes "foo.03", "bar.05.03.01" becomes "bar.05.03.02", etc. The situation where there is more than one index in the requirements ID is described below concerning the requirements for *g_array_free*.

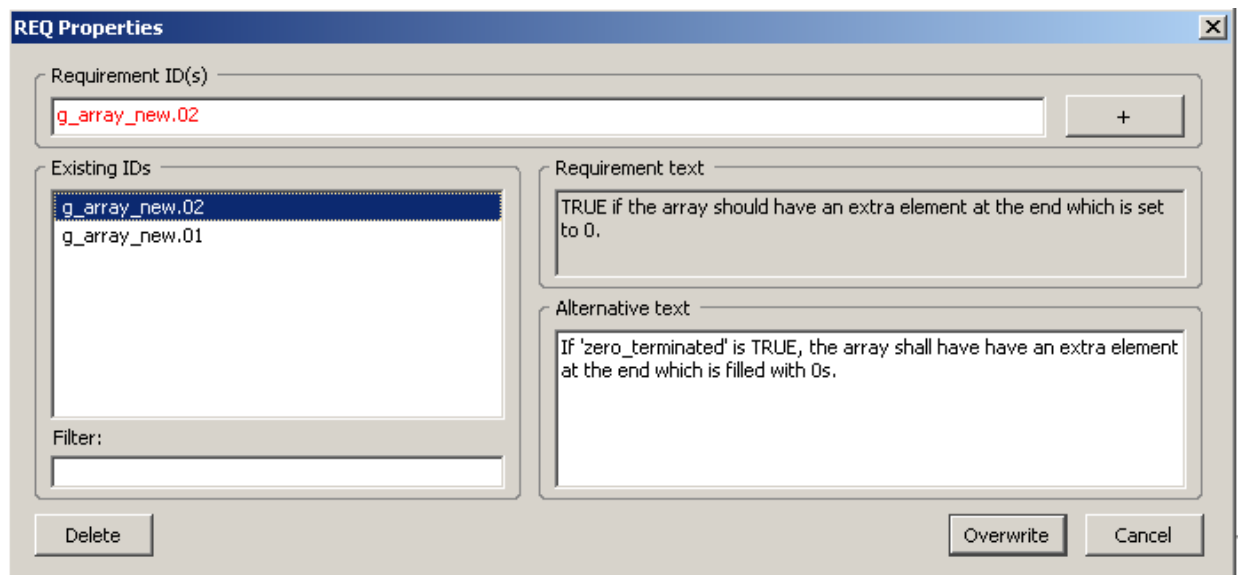
Remember that requirement id should be unique. Within a single html this is enforced by ReqMarkup. The prefixes ("app", "ext", etc.) described below are not taken into account when ReqMarkup checks the IDs for collision (uniqueness violation). For example, the IDs "g_array_new.06" and "ext.g_array_new.06" should not exist in a document at the same time while "g_array_new.05" and "g_array_sized_new.05" do not conflict with each other.

The indices in the ID may not go in order, i.e. it is allowed that there are "foo.03" and "foo.05", but there is no "foo.04".

Press "Add" when the ID is specified.

We added the requirement but forgot to specify its alternative text. To edit an existing requirement, do the following:

1. Do not select any text, just press Ctrl+R. The "REQ Properties" dialog appears again.
2. Select *g_array_new.02* in the list of existing IDs. The "Add" button turns into "Overwrite". Enter the alternative (reformulated) requirement's text.

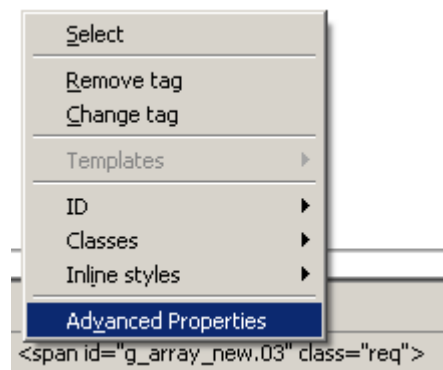


The REQ Properties dialog box is shown. It has a title bar 'REQ Properties' and a close button. The 'Requirement ID(s)' field contains 'g_array_new.02' with a '+' button to its right. Below this is a list of 'Existing IDs' containing 'g_array_new.02' (highlighted) and 'g_array_new.01'. A 'Filter:' text box is below the list. At the bottom left is a 'Delete' button. To the right of the list are two text areas: 'Requirement text' containing 'TRUE if the array should have an extra element at the end which is set to 0.' and 'Alternative text' containing 'If 'zero_terminated' is TRUE, the array shall have have an extra element at the end which is filled with 0s.'. At the bottom right are 'Overwrite' and 'Cancel' buttons.

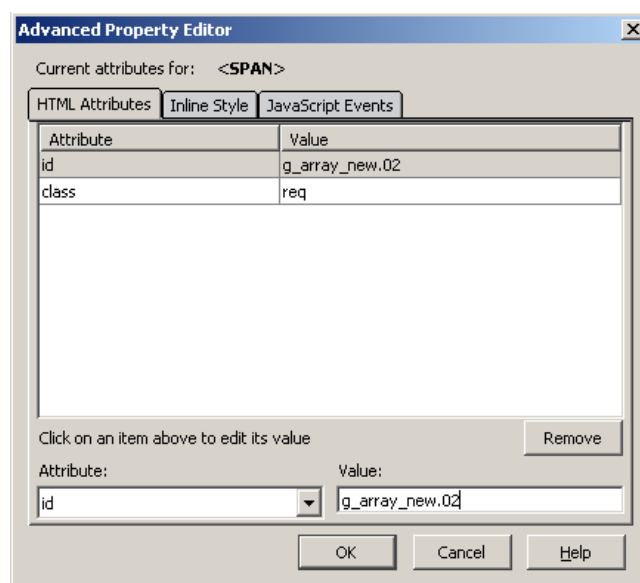
3. Press "Overwrite".

Note that you cannot change requirement ID this way for the current ReqMarkup version. To change the ID without modifying the html source directly, you can do the following in KompoZer:

1. Position the cursor on the marked up text block of the requirement. The KompoZer's status bar will show something like this: ""
2. Right-click this region of the status bar and select "Advanced Properties" from the context menu.



3. Edit the value of the "id" attribute in the dialog, then press "OK".



The Advanced Property Editor dialog box is shown. It has a title bar 'Advanced Property Editor' and a close button. The 'Current attributes for:' field shows ''. There are three tabs: 'HTML Attributes' (selected), 'Inline Style', and 'JavaScript Events'. Below the tabs is a table with two columns: 'Attribute' and 'Value'. The table contains two rows: 'id' with value 'g_array_new.02' and 'class' with value 'req'. Below the table is a text box with the instruction 'Click on an item above to edit its value' and a 'Remove' button. At the bottom, there is a form with 'Attribute:' and 'Value:' labels. The 'Attribute:' dropdown is set to 'id' and the 'Value:' text box contains 'g_array_new.02'. At the bottom are 'OK', 'Cancel', and 'Help' buttons.

The description of the 'clear_' parameter (*"TRUE if GArray elements should be automatically cleared to 0 when they are allocated."*) should be marked up in a similar way as above. The requirement can be reformulated as follows: *"If 'clear_' is TRUE, the GArray elements shall be automatically cleared to 0 when they are allocated."*

g_array_new.03 will be the ID of this requirement.

If you have accidentally added a requirement, you can always remove it using the "REQ Properties" dialog:

1. Press Ctrl+R to open the dialog.
2. Select the requirement you would like to delete in the list of existing requirements. Using a filter (see II.2.3) can be helpful.
3. Press "Delete" button.

You can also remove all requirements marked up in a document by pressing "Remove Markup" button on the toolbar. Beware: this operation cannot be undone.

There seems to be no requirements in the description of the 'element_size' parameter. Leave this description as it is.

There is a requirement in the description of the return value but we have marked up the same requirement before as "g_array_new.01". Still we should mark up this text fragment too but give it a special ID: "&g_array_new.01". The '&' symbol prepended to the ID means that the requirement from this text block has already been marked up somewhere in the same html with ID "g_array_new.01".

This new marked up text block in fact refers to some other requirement in the document (recall C++ references and how '&' is used to denote them - hence the similar notation for the markup).

The referenced requirement may be before or after its "reference" in the document, it does not matter. It should exist in the same document though. References to the requirements from the different documents are not allowed for the present time.

There can be unlimited number of references to a requirement. References to references are prohibited.

From the common sense and from what the documentation says about g_array_sized_new (*"the size of the array is still 0."*), we can assume that the size of the newly created array is 0. That is, the len field of this GArray contains a meaningful value and this value is 0. The documentation says nothing about this, its authors probably considered it obvious. Well, assume it is not so obvious for us.

If we suppose this requirement holds for g_array_new, we can insert new text right in the documentation (the html document) and mark it up. The ID is assigned as usual except it gets the "ext" prefix: ext.g_array_new.06.

The screenshot shows a documentation editor with a table of GArray parameters and a requirement. The requirement text is highlighted with a red box and has its ID, 'ext.g_array_new.06', also highlighted. The table lists parameters: zero_terminated, clear_, and element_size. The editor's toolbar at the bottom shows 'Normal', 'HTML Tags', 'Source', and 'Preview' buttons. The source code view at the bottom shows the HTML structure of the requirement, including the ID 'ext.g_array_new.06'.

zero_terminated:	TRUE if the array should have an extra element at the end which is set to 0.
clear_:	TRUE if GArray elements should be automatically cleared to 0 when they are allocated.
element_size:	the size of each element in bytes

Normal | HTML Tags | Source | Preview

body> <div class="refentry"> <div class="refsect1"> <div class="refsect2"> <p>

"ext" means "extension" in this case. Such requirements with ext-prefixed IDs ("ext-requirements", for brevity) extend the documentation, add something to it that it missed before.

You might wonder how adding an ext-requirement and specifying a reformulated requirement's text differ from each other. The difference is crucial: when we reformulate a requirement, we should not change its meaning. So we should not add any meaning to the requirement it has not had before. We just try to make the requirement's text clearer. If we are rather going to add something new to the documentation, we should use an ext-requirement for that. In practice the line between these cases is sometimes blurred, especially if the requirement's text is somewhat ambiguous.

Checking ext-requirements can be turned on via the T2C configuration parameters. To enable checking of such requirements, you should specify -DCHECK_EXT_REQS in the COMPILER_FLAGS parameter in a generator config file (see IV.C3).

We have finished marking up the requirements for g_array_new. Do not forget to save the html file we have been editing.

Using T2C ReqTools in a browser

The T2C system provides a set of tools that can be useful when the html with the requirement markup is viewed in a web browser. These tools are provided by the ReqTools javascript (**`samples/tools/reqtools/reqtools.js`**). If you open the html files provided for this case study in a text editor, you will notice the line

```
<script language="javascript" src="../../samples/tools/reqtools/reqtools.js"></script>
```

at the beginning of the `<body>`. It is all about that.

Now if you open the html in a browser (currently works with Firefox and Opera, may not work in Konqueror though), you will see a "Show Requirements" link at the top. If you click this link, some other links will appear ("Table >>", etc.) which will be discussed later. Besides that the IDs will now be shown (in curly brackets) before the text of each requirement marked up in this html. This can help to determine if some fragment has been marked up wrong.

Click "Hide Requirements" to return to the original view of the document.

If you click on the link (">>") near the word "Table", a table of all requirements marked up in the document will open. The table shows the text (reformulated, if specified) of the requirements as well as the references to the requirements. You can click the requirement ID to go to the corresponding fragment of the documentation.

The requirement catalog stuff ("REQ Catalogue (XML)>>") will be described later (see II.5).

2.2. `g_array_sized_new`

The requirements for `g_array_sized_new` interface can be marked up in the similar way as above. There are some things you should consider when performing the markup.

There are two requirements for `g_array_sized_new` in the first sentence of its description rather than one as for `g_array_new`. The first one is that the function creates a new array while the second one is that the created array has `reserved_size` elements preallocated. That is, the principle "one sentence – one requirement" is not always true.

There are no requirements for `g_array_sized_new` in the sentence *"This avoids frequent reallocation, if you are going to add many elements to the array."*

2.3. `g_array_append_vals`, `g_array_prepend_vals`, etc.

Now we can markup the requirements for `g_array_append_vals`, `g_array_prepend_vals` and so on up to `g_array_remove_range` inclusive. This should not be a problem.

Note that `g_array_append_val`, `g_array_prepend_val` and `g_array_insert_val` are not LSB-interfaces. So we neither markup nor check the requirements for those unlike `g_array_append_vals`, `g_array_prepend_vals` and `g_array_insert_val`.

When there are many requirements marked up in an html-document, it can be difficult to find the one you need in the list in the "REQ Properties" dialog. The possibility to specify which requirements to show in the list (i.e. to filter them) can prove handy in this case.

A filter is a regular expression and it can be set in the "Filter" the "REQ Properties" dialog. Only those IDs that match the filter will be shown in the list. The empty filter matches any ID.

Often a substring is specified as a filter that should be in the IDs we are interested in (for example, "&g_array_new"). In fact any regular expression can be used here provided its syntax is acceptable by JavaScript. (The syntax is described here for instance: <http://www.javascriptkit.com/jsref/regexp.shtml>)

2.4. `g_array_sort` and `g_array_sort_with_data`

Now let us markup the requirements for the interfaces `g_array_sort` and `g_array_sort_with_data`.

First, note that the description of `g_array_sort_with_data` refers to the description of `g_array_sort`: *"Like g_array_sort(), but <...>".* That is, the requirements for both `g_array_sort` and `g_array_sort_with_data` are specified here at the same time.

In the situations like this several IDs separated by semicolons are specified for the corresponding fragment of the documentation instead of one. For example, "`g_array_sort.01;g_array_sort_with_data.01`"

By the way the text of this requirement should probably be reformulated to explicitly indicate that sorting is done in

ascending order. This is almost obvious, of course, but still.

Second, consider the following fragment: *"which should be a qsort()-style comparison function (returns -1 for first arg is less than second arg, 0 for equal, 1 if first arg is greater than second arg)."*

There is a requirement here too but not for the `g_array_sort` function but rather for any application that uses this function. The application may only pass such a comparison function to `g_array_sort` that meets this requirement. Otherwise nothing is guaranteed.

Requirements like this ("requirements for application", "app-requirements") are often found in the description of input parameters of a function, etc. We need to mark up these too. The rules for assigning IDs are the same as above except they get the **"app"** prefix, for example: `"app.g_array_sort_with_data.03"` `"app.g_array_sort.02;app.g_array_sort_with_data.02"` `"ext.app.my_func.63"`

If an ID has more than one prefix, their order does not matter. I.e. `"app.ext.my_func.63"` and `"ext.app.my_func.63"` is the same ID.

There is another requirement in the description of `g_array_sort_with_data` in the following fragment: *"the comparison function receives a user data argument."*

This fragment means that `g_array_sort_with_data` is implemented so that when the comparison function is called during sorting, its last argument contains the value of `user_data`. So it is a requirement indeed and needs to be marked up.

2.5. `g_array_set_size`

At this stage marking up the requirements for `g_array_set_size` should pose no problem.

2.6. `g_array_free`

When marking up the requirements for `g_array_free`, consider the last sentence: *"Returns : the element data if free_segment is FALSE, otherwise NULL"*

Here several possibilities for the function to behave are described. The compound requirements like this are marked up in the following way.

A so called "parent" requirement is associated with a common part of the text of these requirements (*"Returns:"* in this case). Let us set `"g_array_free.03"` as its ID.

Then two "child" requirements are associated with the following text fragments: *"the element data if free_segment is FALSE"* and *"otherwise NULL"*. These requirements get the IDs `"g_array_free.03.01"` and `"g_array_free.03.02"`, respectively. That is, we simply append one more index to the ID of the parent requirement.

The child requirements can be reformulated as follows: `g_array_free.03.01`: "The function returns the element data if `free_segment` is `FALSE`." `g_array_free.03.02`: "The function returns `NULL` if `free_segment` is `TRUE`."

The parent requirement (`"g_array_free.03"`) is considered checked if and only if all its children have been checked (`g_array_free.03.01` and `g_array_free.03.02`).

Child requirements may have child requirements of their own. The IDs are assigned the same way: for instance the children of `"g_array_free.03.02"` will be `"g_array_free.03.02.01"`, `"g_array_free.03.02.02"`, etc.

2.7. Implicit requirements in the description of `GArray` data fields

Before we finish marking up the requirements, let us take a closer look at the description of `GArray` structure at the beginning of the document. There are two public fields in this structure, `data` and `len`.

The description of these fields imposes some restrictions on the behaviour of `g_array` interfaces.

For example, consider the `len` field: *"guint len; the number of elements in the GArray."* This means that if a function changes the array's length, `len` should change accordingly. That is, say, if we use `g_array_append_vals()` to add 3 elements to an array, `len` should increase by 3 after that.

On the other hand, if a function does not change the number of elements in an array (like `g_array_sort()`), `len` should remain the same.

The implicit requirements like these usually affect all or the most of the interfaces in the particular group. So we should mark them up specifying the lists of ids like the following: `g_array_new.07`; `g_array_sized_new.07`; `g_array_set_size`.

07; ... For the next such requirement we would use `g_array_new.08`; `g_array_sized_new.08`; `g_array_set_size.08`; ...

Using long lists of ids that differ only in the indexes can be tedious and error-prone. There is a more convenient way to handle this: "define"-blocks (see IV.M4).

To create a "define"-block, open the html file that we are marking up in any text editor. Insert the following right after the `<BODY>` tag:

```
<a class="define" name="@g_array_funcs" title="g_array_new; g_array_sized_new;
g_array_set_size; g_array_free; g_array_append_vals; g_array_insert_vals;
g_array_prepend_vals; g_array_remove_index; g_array_remove_index_fast;
g_array_remove_range; g_array_sort; g_array_sort_with_data"></a>
```

It is recommended (but not mandatory) for the names of define-blocks to begin with '@'. This makes it easier to distinguish such names from the ordinary function names, etc.

Now we can use "`@g_array_funcs`" instead of this list of function names in the requirement ids: the description of *data* field will have the id "`@g_array_funcs.07`", while the description of *len* - "`@g_array_funcs.08`".

In order to keep this example (relatively) simple, some of these requirements are not checked in the sample tests we provide.

The ReqTools script (see III.2.1) can also show the define blocks present in the html file. Click on ">>" to the right of the word "defines" at the beginning of the document to see them. (If there are no define blocks in the html, this link will not be shown.)

Markup of the requirements in the "Glib Arrays" documentation is now complete.

3. Generating T2C file template

A T2C file (see II.3) contains a template of the C code for each test case. It is in fact much like "a document with holes" because the placeholders can be left in it to be filled later with the actual test purpose (see part V) parameters. Parameter substitution will be done automatically by the T2C C code generator.

The ReqMarkup plugin can generate a test case template for each interface. To show the template, press "Show/Hide Template" button on the toolbar. The template will be inserted at the beginning of the html file. You can press "Show/Hide Template" again to hide the template.

4. Populating the T2C file template

Overall structure of a T2C file is shown in II.3. Note that the T2C section tags (`<GLOBAL>`, `<BLOCK>`, etc.) should reside on separate lines in the file. Apart from the tags, these lines may contain only whitespace characters.

4.1. The header

Insert a special header at the 1st line of the file (no newlines before are allowed):

```
#library      libglib-2.0
#libsection Arrays
```

Here we specify the name of the library the interfaces under test belong to (`#library`) and the name of the interface group that is tested (`#libsection`).

4.2. Comments

A line beginning with '#' is a comment (and is not considered by the generator) unless '#' is followed by 'library', 'libsection' or a standard C preprocessor directive like 'define', 'undef', 'line', 'ifdef', etc.

Example:

```
# This is a comment in a T2C file.
```

4.3. Globals

#include directives required by the tests should be specified in a <GLOBAL> section that is located right after the T2C file header. The comparison functions for sorting the arrays should also be defined here along with any other global data and functions the tests need. (See **samples/sample-01-t2c/src/glib/glib_arrays.t2c**).

```
<GLOBAL>
#include <glib-2.0/glib.h>

// GCompareFunc
// A comparison function for array elements (necessary for sorting)
gint array_cmp (gconstpointer a, gconstpointer b)
{
    if (a && b)
    {
        if (*((int*)a) < *((int*)b))
        {
            return -1;
        }
        if (*((int*)a) > *((int*)b))
        {
            return 1;
        }
    }
    return 0;
}

// GCompareDataFunc
// A comparison function for array elements that also receives
// user data argument (necessary for sorting)
gint array_cmp_with_data (gconstpointer a, gconstpointer b, gpointer data)
{
    if (a && b && data)
    {
        if (*((int*)a) - *((int*)b) < *((int*)data))
        {
            return -1;
        }
        if (*((int*)a) - *((int*)b) > *((int*)data))
        {
            return 1;
        }
    }
    return 0;
}

</GLOBAL>
```

4.4. Initialization and cleanup of global data

If it is necessary to perform initialization and/or cleanup of some global resources, the respective code should be placed in the <STARTUP> section. Similarly, to clean up these objects place the appropriate code in the <CLEANUP> section.

Example from **samples/sample-02-t2c/src/atk_samples/AtkStreamableContent.t2c**:

```
<GLOBAL>
#include <atk/atk.h>
#include <AtkStreamableContent/MyAtkStreamableContent.h>
#include <useful_functions.h>

AtkStreamableContent* obj = NULL;
</GLOBAL>
```

```

<STARTUP>
    g_type_init();
    OBJECT_NEW(obj, MY_TYPE_ATK_STREAMABLE_CONTENT, "MyAtkStreamableContent");
</STARTUP>

<CLEANUP>
    OBJECT_UNREF(obj);
</CLEANUP>

```

As far as our 1st example ("Glib Arrays") is concerned, we do not need such initialization and cleanup. So it is ok to omit these two sections.

Do not use the <CLEANUP> section to release the resources allocated in the test cases rather than in <STARTUP>. This may cause resource leak. Place such cleanup code in the <FINALLY> subsection of a test case block. See **samples/sample-01-t2c/src/glib/glib_arrays.t2c** for an example that uses <FINALLY>.

Each test purpose is executed in a separate process. Code from <STARTUP> and <CLEANUP> sections is executed in a parent process of the test purpose processes and it is done only once. Each test purpose has its own copy of the global data. So it is pointless to try transferring data between different test purposes via global variables. Ideally, the execution of a test purpose should not affect any other test purpose.

4.5. Writing test case code and specifying parameters

For each interface a test case template block has been generated for you by ReqMarkup in KompoZer (a <BLOCK> section).

Layout of the <BLOCK> section:

```

<TARGETS> ... </TARGETS>
<DEFINE> ... </DEFINE>      // optional
<CODE> ... </CODE>
<FINALLY> ... </FINALLY>    // optional
<PURPOSE> ... </PURPOSE>    // zero or more

```

You should place these subsections in the same order as they are listed above.

The <TARGETS> subsection contains the list of interfaces being tested in this test case, one per line. For example, three interfaces are actually tested in one of the test cases in **samples/sample-01-t2c/src/glib/glib_arrays.t2c**, so the <TARGETS> subsection of the respective <BLOCK> section looks as follows:

```

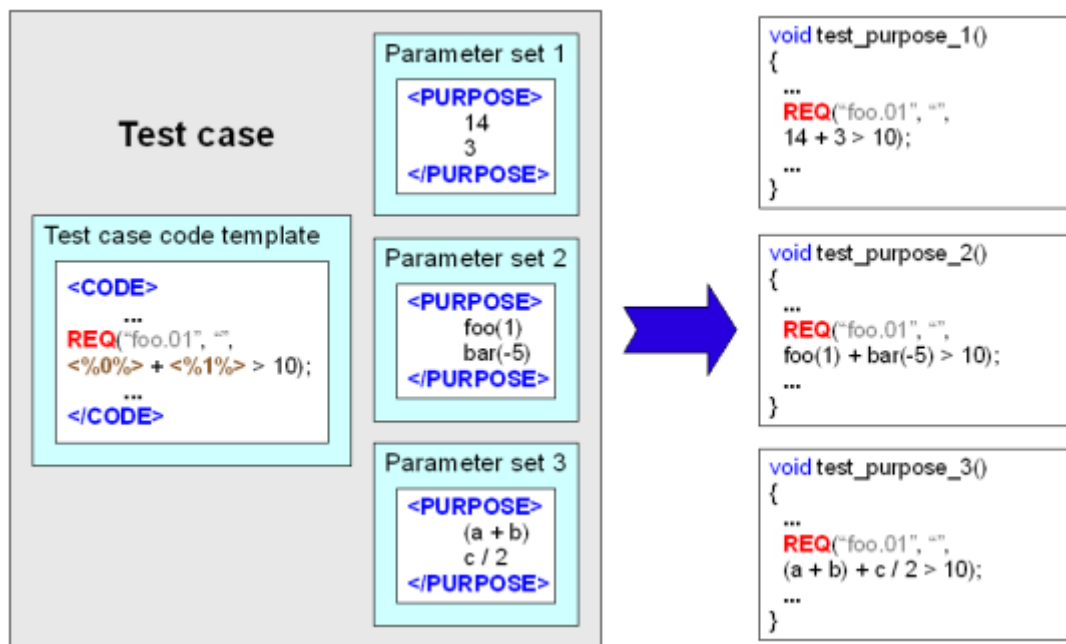
<TARGETS>
    g_array_set_size
    g_array_new
    g_array_sized_new
</TARGETS>

```

The <CODE> subsection contains the common code for each test purpose of this <BLOCK>. The placeholders in the code to be filled by the particular test purpose parameters can be specified here. For example,

```
int nResult = 2 + <%0%> * <%1%>;
```

During the generation of a C function for a test purpose the generator will replace <%0%> and <%1%> with the values from the appropriate <PURPOSE> section.



The `<PURPOSE>` subsection specifies parameters of a particular test purpose (one parameter per line). The first of these parameters will replace `<%0%>` in the `<CODE>` section, the next one will be for `<%1%>` and so on. Up to 256 parameters are allowed (`<%0%>` - `<%255%>`).

Example:

```
<PURPOSE>
    2
    a+b
</PURPOSE>
```

`<PURPOSE>` subsection is optional. If no parameters are necessary for a test purpose, you may omit this section or specify it but leave it empty, like this:

```
<PURPOSE>
</PURPOSE>
```

Multiple `<PURPOSE>` sections are allowed in a `<BLOCK>`.

For each `<PURPOSE>` subsection a single C function (in fact, a test purpose - see part V) will be generated using the template code specified in the `<CODE>` section.

In the `<DEFINE>` section you can list the `#define` directives for the test purpose parameters. The code generator will place these directives at the beginning of the generated test purpose function. Corresponding `#undef`-directives will be inserted at the end of this function.

This feature can be used to replace `<%...%>` with more readable symbolic names which can be quite convenient. `<DEFINE>` subsection is optional.

Example:

```
<DEFINE>
#define QUANTITY <%0%>
#define PRICE    <%1%>
</DEFINE>
```

Now instead

```
int nResult = 2 + <%0%> * <%1%>;
```

we can write the following in the `<CODE>` subsection:

```
int nResult = 2 + QUANTITY * PRICE;
```

The code from the <FINALLY> subsection is always executed in the test purpose regardless of whether the requirement checks in REQs pass or fail. You can use this subsection to release the resources local to the test purpose, e.g. free previously allocated memory, close files, etc. (To release global test case resources, use the <CLEANUP> section described above.) The <FINALLY> subsection is optional.

The complete example of a <BLOCK> section with all its subsections is shown below. Usage of REQ(), TRACE() and other macros is explained later.

Example:

```
<BLOCK>

<TARGETS>
    g_array_remove_index_fast
</TARGETS>

<DEFINE>
#define INDEX        <%0%>
#define VALS         <%1%>
#define TYPE         <%2%>
</DEFINE>

<CODE>
    GArray *ga = NULL;
    GArray *new_ga = NULL;
    int old_len;
    int last_el;
    TYPE vals[] = VALS;

    ga = g_array_new(FALSE, TRUE, sizeof(TYPE));
    if (ga == NULL)
    {
        ABORT_TEST_PURPOSE("g_array_new() returned NULL.");
    }

    ga = g_array_append_vals(ga, vals, sizeof(vals) / sizeof(TYPE));
    if (ga == NULL)
    {
        ABORT_TEST_PURPOSE("g_array_append_vals() returned NULL.");
    }

    old_len = ga->len;
    last_el = g_array_index(ga, TYPE, old_len - 1);

    new_ga = g_array_remove_index_fast(ga, INDEX);

    /*
     * the GArray.
     *
     * [The function returns a pointer to the modified GArray.]
     */
    REQ("g_array_remove_index_fast.03", "g_array_remove_index_fast returned
    NULL", new_ga);
    REQ("g_array_remove_index_fast.03",
        "The returned GArray pointer does not match the original one.",
        new_ga == ga);

    TRACE("The length of the array is %d (should be %d).", ga->len, old_len -
1);

    /*
     * Removes the element at the given index from a GArray.
     */
```

```

    REQ("g_array_remove_index_fast.01;g_array_remove_index_fast.08", "", ga->len
= old_len - 1);

    if (INDEX < old_len)
    {
        /*
        * The last element in the array is used to fill in the space
        */

        REQ("g_array_remove_index_fast.02",
            "The last element of the array did not fill in the space.",
            g_array_index(ga, TYPE, INDEX) == last_el);
    }

</CODE>
<FINALLY>
    if (ga)
    {
        g_array_free(ga, TRUE);
    }
</FINALLY>

<PURPOSE>
    8
    {19, 89, -1, 8, 7, 190, 9, 10, 28, 56}
    int
</PURPOSE>

<PURPOSE>
    0
    {19, 89, -1, 8, 7, 190, 9, 10, 28, 56}
    int
</PURPOSE>

<PURPOSE>
    9
    {19, 89, -1, 8, 7, 190, 9, 10, 28, 56}
    int
</PURPOSE>

</BLOCK>

```

4.6. REQ and other useful macros

The <CODE> subsection contains at least one REQ() call for each marked up requirement for the particular interface.

4.6.1. REQ

REQ macro is used to check requirements and report failures. Syntax:

```
REQ(<list_of_IDs>, <comment>, <expression>);
```

Example:

```
REQ("fake.01;foo.04",
    "Incorrect multiplication result",
    ARG0 * ARG1 == nCorrect);
```

If <expression> is nonzero, the execution of the test purpose goes on. A message is also output to the TET journal

indicating that the requirements with IDs listed in the 1st REQ argument have been checked.

Otherwise, i.e. if the requirement is violated, another kind of message is output to the TET journal that contains the list of the requirement IDs, the text of these requirements and also the comment specified as the second REQ argument. The execution of the test purpose is aborted in this case and the result code is set to FAIL.

If execution of a test purpose has not been aborted due to a failed requirement check or some unexpected failure (for example, segmentation fault, glibc error, etc.), the test result code is set to PASS.

Note that

```
REQ(<list_of_IDs>, <comment>, <expression>);
```

is equivalent to

```
TRACE0(<comment>);  
REQ(<list_of_IDs>, "", <expression>);
```

(See the description of TRACE0 below.)

You can use TODO_REQ() macro as the <expression> parameter for the requirements, checks for which are yet to be written. In this case no record goes to the TET journal and the execution of the test goes on as if the check has passed.

Unlike this, REQs with TRUE (or 1) as the expression do add records to the TET journal as described above, despite such checks never fail. The REQs like these can be used to report that the requirement is covered in the test even if it is satisfied automatically due to the way the test is organized.

If the REQ fails, the subsequent REQs in this test purpose WILL NOT be checked. The code in the <FINALLY> subsection (if present) will be executed and then the test purpose will terminate.

Sometimes more than one requirement is actually checked in a single REQ. It can happen that we are unable to determine which requirement has failed when the <expression> evaluates to FALSE (i.e. 0). This is often the case for get_XXX() and set_XXX() functions: we can often check that set_XXX() has worked as needed only by calling get_XXX() and then comparing the value it has returned with the value we tried to set by set_XXX. If the values do not match we cannot really say whether it was get_XXX() or set_XXX() (or both) that went wrong.

We can specify the list of corresponding requirement IDs in the 1st parameter of REQ in situations like this.

Example:

```
/*  
 * If both key and group_name are NULL, then comment will be written  
 * above the first group in the file.  
 */  
TRACE("set_comment() was called for \"%s\\", get_comment() returned \"%s\\".",  
      COMMENT, ret_cmnt);  
REQ("g_key_file_set_comment.03;g_key_file_get_comment.03",  
    "",  
    is_comment_equal(COMMENT, ret_cmnt));
```

If the expression in this REQ is false the message output to the journal will say that at least one of the listed requirements has failed.

Do not confuse the list of IDs in a REQ in situations like this with the list of IDs we specified during the markup (see the markup of the documentation for g_array_sort() in III.2.4). These lists serve completely different purposes. In the latter case a REQ() call is generated by ReqMarkup for each requirement in the list, so that the ID list for each REQ() still contains one and only one ID.

If the failed requirement is a requirement for the application using the interface being tested (i.e. a requirement with an "app."-prefixed ID) the displayed message indicates that there can be a bug in the test case itself (perhaps a test case developer error).

4.6.2. TRACE and TRACE0 - trace message output to the TET journal and/or stderr

You should use TRACE instead of printf (the syntax is the same):

```
TRACE("The length of the array is %d (should be %d).", ga->len, old_len -
```

```
1);
```

instead of

```
printf("The length of the array is %d (should be %d).", ga->len, old_len - 1);
```

TRACE0 should be used instead of calling printf() with the only argument:

```
TRACE0(str);
```

rather than

```
printf(str);
```

4.6.3. RETURN

This macro sets the result code to PASS (if no failure has occurred before in this test) and ends the test purpose.

Code in the <FINALLY> section will be executed anyway.

4.6.4. INIT_FAILED(<message>)

One can encounter a situation when an error is found during the execution of the startup function, and it makes no sense to execute the test case after that. (For instance, the initialization may have failed for some global user-defined object.)

In this case you should call INIT_FAILED(...) in the <STARTUP> section providing appropriate description of the failure as its argument. This description will be written to the TET journal. No test purpose will be executed after that for the test cases from this t2c-file. All the test purposes will be marked as UNINITIATED.

Example:

```
<STARTUP>
g_type_init ();

img = ATK_IMAGE (g_object_new (TEST_TYPE_SIMPLE_IMAGE, NULL));
if (!img)
{
    INIT_FAILED("Unable to create a TestSimpleImage instance.");
}
</STARTUP>
```

4.6.5. ABORT_TEST_PURPOSE(<message>)

Use this macro to abort test purpose execution if something wrong happens (e.g. memory allocation failed for some local data, etc.) The specified message will go to the TET journal, the test purpose result will be set to UNRESOLVED and (after the execution of the <FINALLY> section) the test purpose terminates.

Example:

```
ga = g_array_new(FALSE, TRUE, SIZE);
if (ga == NULL)
{
    ABORT_TEST_PURPOSE("g_array_new() returned NULL.");
}
```

4.6.6. ABORT_UNSUPPORTED(<message>)

Use this macro in the test purpose to abort execution if the feature to be checked is not supported by the system under test. The specified message goes to the journal and should describe the situation, e.g. "Dynamic loading of modules is not supported". The result of this test purpose will be set to UNSUPPORTED.

Code in the <FINALLY> section will be executed anyway.

All the macros described above are defined in <t2c.h>.

4.7. Sample T2C files

The t2c files for the examples described here can be found in **samples/sample-01-t2c/src/glib/**, **samples/sample-02-t2c/src/atk_samples/**. The tests in these files may not be perfect, of course, (and there are also some TODO_REQ()s there) but still they can be useful.

5. Creating requirement catalogue

For each t2c-file there should be an xml-file with the same name in the "reqs" subdirectories of **samples/sample-01-t2c/** and **samples/sample-02-t2c/**. Each of the files should contain a list of the requirements (with their texts) to be checked in the tests from the t2c file and is in fact a requirement catalogue for a given interface group.

This catalogue will be used by the tests in failure reporting. It contains the ID and the text (original or reformulated, if specified) for each requirement.

The catalogue is created by the ReqTools javascript (see III.2.1) in a web browser. (For the present, it has been tested for Mozilla Firefox, Opera and Internet Explorer. It may not work properly in Konqueror though.)

Let us create a requirement catalogue for "Glib Arrays" group.

Open the marked up html in a browser. You will see a "Show Requirements" link at the top. (If you don't, check if reqtools.js is specified in a proper <script> tag in the html. See the details [here](#).)

Click this link. Some other links will appear. Among these there should be "REQ Catalogue (XML)>>". Click on ">>" and you will see something like this:

```
<?xml version="1.0"?>
<requirements>
<req id="g_array_append_vals.01">
    Adds len elements onto the end of the array.
</req>
<req id="g_array_append_vals.02">
    The function returns a pointer to the modified GArray.
</req>
<req id="g_array_free.01">
    Frees the memory allocated for the GArray.
</req>
<req id="g_array_free.02">
    If free_segment is TRUE it frees the actual element data as well.
</req>
<req id="g_array_free.03.01">
    The function returns the element data if free_segment is FALSE.
</req>
<req id="g_array_free.03.02">
    The function returns NULL if free_segment is TRUE.
</req>
...
</requirements>
```

The requirement ID is specified in the "id" attribute of the "req" tag. The contents of this tag (the text between ">" and "</req>") are the requirement's text or the alternate requirement's text if it was specified.

Copy the displayed text to **samples/sample-01-t2c/reqs/glib_arrays.xml**. The requirement catalogue for "Glib Arrays" is ready. Now you can create a requirement catalogue for "AtkStreamableContent" interface group from our second example.

Sometimes the requirements for an interface group are specified in several html files rather than one. In this case the complete requirement catalogue for this group should be assembled manually from the parts generated by the ReqTools script for each of these html files.

6. Generating C code and building the tests

If the tests are organized in appropriate directory structure described above (see III.1), you can just change current directory to **samples/** and run **build_all.sh** from there to generate the C code and build the tests.

build_all.sh first sets some necessary environment variables (T2C_ROOT, PATH, PKG_CONFIG_PATH, etc.) then builds the T2C code generator. The generator is then used to create C code from T2C files and makefiles to build test executables from these C sources. Finally, these makefiles are executed along with the makefiles in the testdata_src subdirs (if present).

The command-line syntax of the T2C code generator is described in IV.C2. You can look at the **gen_tests** scripts found in each subsuite directory to see which parameters are actually used for T2C to process the presented samples.

7. Executing and debugging the tests

7.1 Executing the tests

To run all the tests, execute the **run_tests.sh** script from the **samples/** directory.

If you want to run the tests from a particular "suite" (sample-01-t2c or sample-02-t2c), specify the name of this suite as a parameter for **run_tests.sh**.

Example:

```
./run_tests.sh sample-01-t2c
```

In fact, **run_tests.sh** performs some auxiliary operations and then runs the TET test case controller from the **samples/** directory: `tcc -e .`

The test results can be found in the TET journal (**samples/results/0001e/journal**).

7.2 Debugging the standalone ("no-TET") version of a test

A standalone version of a test can also be built from the generated C source and executed without TET environment. This can be useful, for example, to debug the test and the target system or to obtain more data about the behaviour of the tested interfaces in case of failure.

The instructions on how to build and execute a standalone version of the test are available in IV.D1. A brief description is also provided in a comment block in the beginning of each generated C file.

IV. Notes

M1. KompoZer Ctrl+R fix

The T2C ReqMarkup plugin uses Ctrl+R as a default keyboard shortcut to bring up the "Requirement properties" dialog. However, this shortcut is already assigned to "filerevert" command in KompoZer 0.7.7. You could use Alt+R instead or circumvent this problem by changing a shortcut for "filerevert". In the latter case you can do the following:

1. Unpack {**KompoZer_install_directory**}/chrome/comm.jar to some temporary directory, say, /tmp/kz (this jar is actually an ordinary zip-archive).
2. Open /tmp/kz/content/editor/editor.xul in your favourite text editor.
3. Find the following text: `<key id="revertkb" key="&filerevert.keybinding;" observes="cmd_revert" modifiers="accel"/>`
4. Replace 'modifiers="accel"' with 'modifiers="accel,shift"' (thus the "filerevert" command will be assigned Shift+Ctrl+R shortcut key, leaving Ctrl+R open for our assignment.).
5. Save the file.
6. Pack /tmp/kz/content directory in a zip archive named **comm.jar**.
7. Replace the original **comm.jar** with this new one.
8. Restart KompoZer.

Ctrl+R will now work properly.

M2. ReqMarkup installation

To install the T2C ReqMarkup plugin on KompoZer:

1. Start KompoZer.
2. Open "Tools" -> "Extensions" in the main menu.
3. Press "Install" and select **reqmarkup-X.Y.Z.xpi** (X.Y.Z is a version number) from **samples/tools/reqmarkup/**.
4. Press "Install" and then restart KompoZer.

M3. Requirement IDs for properties and signals (Glib Object System)

Sometimes it can be convenient not to link the requirements for properties and signals of gobject-based objects (see <http://www.gtk.org/api/2.6/gobject/>) to the interfaces which behaviour they affect. (In fact the problem is that it is often difficult to determine which interfaces are affected by the requirements for a given property or a signal.)

For simplicity we can assign IDs for these requirements using almost the same rules as for the functions. Name of the property or signal should be used in the ID instead of the function name and dashes ('-') should be replaced with underscores ('_'). It is recommended to prefix the name of the property (or signal) with a name of the class this property (or signal) belongs to. This can help to avoid ID conflicts if there are properties or signals with the same names in different classes. The name of the class should be written in lowercase with underscores separating the words.

Examples:

ext.atk_object.accessible_parent.05

atk_object.accessible_table_column_header.01.03

atk_document.state_change.02

M4. Substitutions in the IDs ("define"-blocks for long lists of IDs)

If a fragment of the documentation contains a requirement for many interfaces, it can be difficult to specify all respective IDs in the list correctly. For instance, the following ID list could be used for one of the fragments of the description of printf():

"asprintf.13.02;fprintf.13.02;printf.13.02;snprintf.13.02;sprintf.13.02;vasprintf.13.02;vdprintf.13.02;vfprintf.13.02;vprintf.13.02;vsnprintf.13.02;vsprintf.13.02"

Imagine you need to write such a list for each requirement trying not to forget to change the indexes appropriately. It can be a really hard time.

There is a way to handle this. We can create a so called "define"-block; (or "substitution block"): denote the whole list of functions (sprintf, fprintf, printf, snprintf, sprintf, vasprintf, vdprintf, vfprintf, vprintf, vsnprintf, vsprintf) with a single identifier, say, @print_funcs. (It begins with '@' to distinguish it from function names. This is not mandatory though.) Its definition should be placed in the html we are marking up and it looks like this:

```
<a class="define" name="@print_funcs" title="asprintf; sprintf; fprintf; printf; snprintf; vasprintf; vdprintf; vfprintf; vprintf; vsnprintf; vsprintf"></a>.
```

Now instead of that long list we can write just this: "@print_funcs.13.02".

For the present the define-blocks should be placed in the html manually. It is recommended that these blocks were in the beginning of the body of this html.

The prefixes ("app", "ext" etc.) work properly with the substitutions. That is, the following "ID lists" are allowed: "ext.@print_funcs.10" "app.@print_funcs.15".

A substitution identifier should consist of letters, may contain digits, '_' and '@'. Other symbols are not allowed. The identifier is case sensitive.

The substitution ID may be an element of another ID list like any ordinary ID. Moreover, it is possible to use substitutions in a definition of another substitution.

Example: ``, f_PRINT_FUNCS, @_PRINT_FUNCS, s_PRINT_FUNCS are defined somewhere else in this html (no matter before or later). Please do not create recursion here.

References to substitution identifiers (like "&@print_funcs.13.02") are not allowed in the current version of ReqMarkup.

C1. Directory structure of a T2C test suite

The root of this directory structure is assumed to be specified in the T2C_ROOT environment variable.

The subdirectories of **src** (denoted below as <s1> and <s2>) are just used to group the T2C files. Often there is only one subdirectory in **src** that contains all the T2C files.

```
T2C_ROOT
|
+-- tools
|   |
|   +-- t2c                // T2C code generator
|
+-- <main_suite_name>-t2c/ // the test suite
|   |
|   +-- <subsystem>-t2c/   // a "subsuite" containing tests for a given library
// (like glib, fontconfig, ...)
|       |
|       +-- include/      // common include files for the tests for this
subsystem
|           |
|           +-- reqs/      // requirement catalogue
|               |
|               +-- <group1>.xml
|               +-- <group2>.xml
|               +-- <group3>.xml
|               ...
|           +-- src/       // T2C files
|               |
|               +-- <s1>/  // This directory contains a group of T2C files
|                   |
|                   +-- <group1>.t2c
|                   +-- <group2>.t2c
|                   |
|               +-- <s2>/
|                   |
|                   ...
|           +-- scenarios/ // TET scenarios
|               |
|               +-- func_scen
|
+-- tests/              // Generated C-files are placed here
|   |
|   +-- <group1>/
|       |
|       +-- <group1>.c
|       |
|       +-- <group2>/
|           |
|           +-- <group2>.c
|           ...
+-- testdata/           // Data used by the tests should be stored here
|   |
|   +-- testdata_src/   // Contains a makefile and source code
// of the test data that needs to be built
(modules, etc.)
|       |
|       +-- <group1>/
|           |
|           ...
|       +-- <group2>/
```

```

|   |   |
|   |   ...
|   ...
...

```

C2. Command line syntax of the T2C code generator

Synopsis:

```
t2c <main_dir> <test_dir> [cfg_path]
```

The paths are relative to the directory specified in \$T2C_ROOT environment variable.

main_dir - main directory for the test suites (<main_suite_name>-t2c directory in a structure shown in IV.C1 above). The subdirectories of <main_dir> contain the test "subsuites" for the particular subsystems to be tested. It is often the same directory as specified in the T2C_ROOT environment variable.

If this file does not exist when the code generator is invoked, it will be created. Otherwise an appropriate record will be added to the existing file.

tetexec.cfg should also reside in this directory. (This file is not generated automatically in this T2C version.)

test_dir - path to the directory of a test suite to be generated. Example: **sample-01-t2c**.

The C-code generator looks for the t2c files in the **src** subdir of this directory. The generated C files will be placed in the **tests** subdir, local scenario file(s) - in **scenarios**. For the above example these directories are the following (assume \$T2C_ROOT is **samples/**): sample-01-t2c/src, sample-01-t2c/tests, sample-01-t2c/scenarios.

cfg_path - (optional) path to the configuration file of the generator (see IV.C3). If this path is not specified, default configuration will be used.

Example: (this command is executed from the **samples** directory)

```
tools/t2c/bin/t2c . sample-01-t2c sample-01-t2c/sample-01.cfg
```

C3. Configuration parameters of the T2C C code generator.

The parameters can be specified in a configuration file of the code generator as follows:

<NAME>=<value> Example:

```

COMPILER=lsbcc
COMPILER_FLAGS=`pkg-config --cflags glib-2.0` -DCHECK_EXT_REQS
LINKER_FLAGS=`pkg-config --libs glib-2.0`
TET_SCEN_RECORD=no
WAIT_TIME=180

```

The parameters can be listed in any order, one per line. The empty lines are ignored.

COMPILER - the compiler to be used to build the generated tests. The value of this parameter (as well as **COMPILER_FLAGS** and **LINKER_FLAGS**) goes to the common test makefile, **common.mk**.

COMPILER_FLAGS and **LINKER_FLAGS** - additional compiler and linker options required to build the tests. These flags will be copied to the makefiles of these tests.

The contents of "\$T2C_ROOT/cc_special_flags" (if this file exists) will also be loaded as the additional compiler flags in the generated makefiles. The file usually contains the version-specific compiler flags (e.g., -fno-stack-protector for gcc 4.1.0 and newer.). This file is often absent or is empty. You should only fill it when you really have to.

TET_SCEN_RECORD - if the value of this parameter is 'yes' (or 'YES'), the generator will add a proper record in the main TET scenario file (tet_scen) after the code of the test is created. This record may look like this:

```
:include:/sample-t2c/scenarios/func_scen
```

If this parameter has any other value (e.g. 'no'), no record is added in tet_scen.

WAIT_TIME - if positive, specifies how long (in seconds) any single test purpose is allowed to run (the 'controller' process will wait no longer than that, see appendix 1, item 1.5 for details). If this parameter has zero or negative value, there will be no restrictions on test purpose execution time.

Default value: 30.

It is highly recommended that a reasonable positive value is specified for WAIT_TIME. This ensures that the test suite execution will be finished even if some of the tests hang.

You should probably set WAIT_TIME to 0 to debug a standalone version of a test.

C4. Access to test data directories from the code

One often needs to use external data to check some of the requirements. For example, the test will probably have to load data from a file.

It is recommended to place the data necessary for the tests in the subdirectories of the **testdata** directory for the test suite. These subdirectories should have the same names as the corresponding t2c files. So the tests from a t2c file will look for their data in a separate directory.

To obtain the path to these data from the test code, one can use the `T2C_GET_DATA_PATH(rel_path)` macro.

Suppose we need to get the path to the file **myfile.txt** from the test which source is in **glib_key_parser.t2c**. Let **gkp-t2c** to be the path to the test suite subdirectory (relative to `$T2C_ROOT`), `T2C_ROOT = /tmp/test`.

In this case `T2C_GET_DATA_PATH("myfile.txt")` will return `"/tmp/test/gkp-t2c/testdata/glib_key_parser/myfile.txt"`.

The returned pointer to the string should be freed when it is no longer needed.

Example (from **samples/sample-02-t2c/include/AtkStreamableContent/AtkStreamableContent.h**)

```
GError *error = NULL;
char* full_filename = T2C_GET_DATA_PATH(file_names[i]);
GIOChannel* channel = g_io_channel_new_file(full_filename, "r", &error);
if(error != NULL)
{
    ...
}
free(full_filename);
return channel;
```

C5. Common directory for header files

Recall that the t2c-sources of the tests reside in **samples/sample-02-t2c/src** while the generated C sources will be placed in **samples/sample-02-t2c/tests**. Suppose we need to write some header files for these tests. The question is where we should place these headers. The common include directory is provided exactly for this purpose.

The 2nd example (**sample-02-t2c**) demonstrates how this directory can be used. `$(T2C_ROOT)/sample-02-t2c/include` is automatically specified in -I compiler option in a make file created by the T2C code generator. So when we need to include the headers it contains, we just write :

```
#include <AtkStreamableContent/MyAtkStreamableContent.h>
#include <useful_functions.h>
```

(See **samples/sample-02-t2c/src/AtkStreamableContent/AtkStreamableContent.t2c**.)

D1. Standalone test execution and debugging

For each subsystem to be tested, T2C generates not only required C sources but also makefiles. Among these is **common.mk** that can be found in `$T2C_ROOT/<test_dir>/tests/<test_name>/.mk`. (This .mk-file contains common definitions for building the tests and it is included in the makefiles for each test. The latter makefiles contain targets to build the test for execution under TET ("all") and for standalone execution and debugging ("debug").

To build a standalone version of the test, you can do the following.

1. Make sure the 'T2C_ROOT' environment variable is properly defined, T2C is built and C source for the .t2c-file of interest has already been generated.
2. Change current directory to **\$T2C_ROOT/<test_dir>/tests/<testname>** Example:

```
cd $T2C_ROOT/desktop-t2c/gmodule-t2c/tests/gmodule
```
3. Type "make clean" and then "make debug" in the command line. This will build the standalone version of the test.

During the build process the compiler will use the **tet_api.h** file we provide (it can be found in **\$T2C_ROOT/tools/t2c/debug/include**) instead of the one from TET. The object file for the test will be linked with **dbg.o** and **t2c_util_d.a** from **\$T2C_ROOT/tools/t2c/debug/lib** instead of **tcm.o** and **t2c_util.a**. If you want, for example, to debug the test in an Eclipse project, you should specify this #include path and these linker input files in the build settings of the project.

Only a subset of TET API is supported in the debug components. Avoid using TET API directly from the t2c-file. Instead use special macros defined by T2C (REQ(), TRACE(), ABORT_TEST_PURPOSE() etc).

Now the test can be executed. Syntax:

```
./<testname> [-v] [IC_number]
```

-v If this option is specified, the test will be executed in a verbose mode. Note that value of the TET variable named "VERBOSE" has no effect on the standalone test execution. TET settings have nothing to do with this, anyway.

IC_number It is a number of the invocable component to execute. Each TET compliant test contains at least one invocable component (IC). (The number of the IC is specified as the 2nd field of the tet_testlist structure.) Each invocable component consists of one or more test purposes. In the C source of the test each element of the tet_testlist array is a pair of the test purpose name and the IC number. See TET documentation for details.

If 'IC_number' is not specified, all invocable components will be executed.

Typically, there shall be only one test purpose for any invocable component generated by T2C. So IC number will be unique for each test purpose in the C file. You may change IC numbers in the tet_testlist array as you like. You can, for example, give several test purposes the same IC number, say, 999. Then execute

```
./<testname> 999
```

All these test purposes will be executed. Do not change test purpose names listed in this array.

There is no TET journal for standalone test execution. All messages from the test go to stderr if the verbose mode is on or to nowhere if it is off. tet_printf() does nothing in both these cases, you should use TRACE() and TRACE0() instead.

Examples:

```
./gmodule
```

(Runs all the invocable components defined in this executable. Verbose mode is off.)

```
./FcCharSet -v 17
```

(Turns verbose mode on and runs all the test purposes in the invocable component number 17.)

To run the test under TET again, just rebuild this test:

```
make clean
make
```

Because each test is executed in a separate process, it may be necessary to let the debugger know this. Some of the debuggers follow the parent process by default, which is probably not what you want. As for gdb, the following gdb command will handle this problem:

```
set follow-fork-mode child
```

If you want to debug a standalone version of the test, make sure the WAIT_TIME configuration parameter (see IV.C3) is set to zero. If WAIT_TIME is positive, the test may be terminated in the middle of debugging because its time has

expired.

Alternatively, you may set the wait time to 0 in the C-file itself and leave WAIT_TIME alone.

To do this, find a call of `t2c_fork()` in the `tp_launcher()` function. The 3rd argument passed to `t2c_fork()` is the wait time, that is where the WAIT_TIME value goes when the C-file is generated. Change this argument to 0, rebuild this test only and then you can debug it.

V. Glossary

Note that the definitions below may not be strict. It is quite difficult to provide strict and yet clear definitions for some of the terms described here.

"Shallow"-quality tests

These are usually simple tests with the only guaranteed purpose: to check that the interface does not crash being called with some set of correct parameters and in the correct environment. Additionally it can be checked that the interface does not return an error code.

"Normal"-quality tests

These tests check the main functionality of an interface and may check a few error scenarios.

"Deep"-quality tests

Deep tests check the most of the specification assertions in various conditions/states.

Elementary requirement (elementary assertion)

Elementary requirement is a requirement (assertion) that cannot be split into smaller requirements or it is not reasonable to do so.

Here we only consider the requirements concerning the **behaviour** of the interfaces to be tested. That is, *"The function returns 777 on success"* is a requirement of this kind while *"The structure contains the following fields: int num; double dub; <...>"* is not.

Test purpose, TP

From TETware User Guide (<http://tetworks.opengroup.org/documents/3.7/uguide.pdf>):

A test purpose typically tests an individual element of system operation for conformance to some statement of required behaviour, and yields a result indicating whether or not the element passed the test.

As far as T2C is concerned,

Test purpose = {code from the <CODE> section} + {a set of parameters from the <PURPOSE> section}

A test purpose corresponds to a function in the generated C file ("test purpose function"). Each test purpose function in a particular C file has a unique number that is shown in the name of the function, e.g. "test_purpose_4", "test_purpose_55", etc.

Ideally, the test purposes should be independent on one another. In T2C each test purpose function is executed in a separate process and has its own copy of global data.

A test case consists of one or more test purposes that check a common set of requirements (assertions) for the same interface(s). Each <BLOCK> section in a T2C file (see II.3) corresponds to a test case.

From TETware User Guide:

A test suite is the largest grouping of tests that can be processed by the TETware Test Case Controller. A test suite is

made up of one or more test cases. A test case is the smallest test program unit that can be built or cleaned up by the Test Case Controller. A test case consists of one or more invocable components. An invocable component is the smallest test program unit that can be executed by the Test Case Controller.

In our case each invocable component contains a single test purpose.