GreenConfig User's Guide (GreenConfig v.4.3.0)

Developed by
Christian Schröder and Wolfgang Klingauf
Technical University of Braunschweig, Dept. E.I.S.

10^{th} May 2011

Contents

Chapter 1

Introduction

The GREENCONFIG project aims to provide a library that can be used to connect User defined configuration mechanisms to Tool configuration mechanisms, giving both flexibility.

The diagram 1.1 shows an IP block connected, via the configuration library, to an ESL tool that is providing configuration from a database.

The IP block has one API to the library. The library uses a different API to the ESL tool.

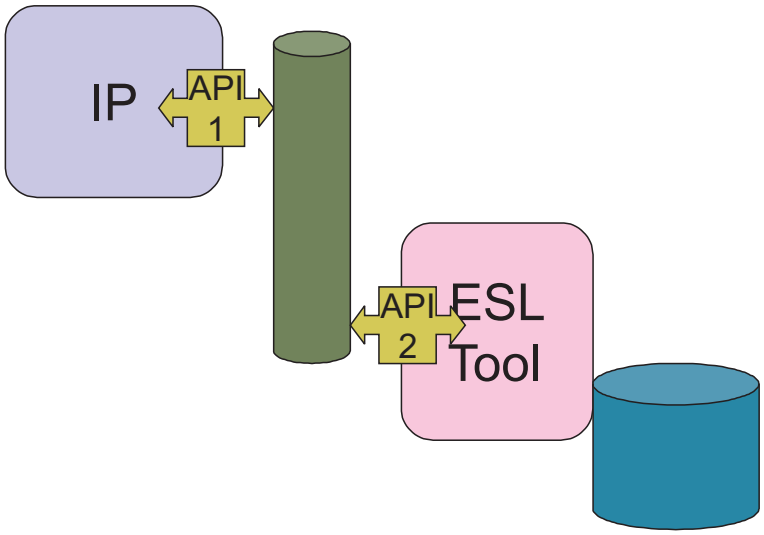

Figure 1.1: IP block, ESL tool and GreenConfig Library

This allows IP designers to choose the configuration mechanism that best suits their style of IP creation, while staying independent of tools.

1.1 GreenControl Middleware

See the GREENCONTROL User's Guide and web page[1] for a description of the GREENCONTROL framework which is the base of GREENCONFIG.

[1] GREENCONTROL project page: http://www.greensocs.com/projects/GreenControl

1.1.1 User APIs

Config APIs Config APIs are *special* GREENCONTROL *User APIs* whose task is the parameter configuration (GREENCONFIG) and communicate with the *ConfigPlugin*.

- Different API calls are translated into transactions with different command fields performing the actions inside the plugin.
- Config APIs receive transactions from the ConfigPlugin and process them, e.g. notification of value change events.

Config API implementations should be kept minimalistic. Each parameter access leads to a transaction to the ConfigPlugin. The other way around the developer of an API may want to cache parameter values locally.

There are different types of configuration User APIs:

- GreenConfig parameters (gs_param) are the default native objects that should be used to make a model configurable.
- The GreenConfig API (GCnf_Api) is the native API to be used by any configurating unit (e.g. testbenches, tools, models configuring other models)
- Several adapter APIs allow integration of other configuration mechanisms or models using those.

1.2 ConfigPlugin

The ConfigPlugin is a GreenControl Service Plugin. It provides all functionality required by the Config APIs (cp. requirement list in appendix A.1).

1.3 Parameter management

The configurable parameters are managed by the ConfigPlugin. To load and store parameters, it uses a Param-API which is connected to a storage implementation. This might be

- a simple map
- an SQL database
- ...

Chapter 2

GreenConfig

2.1 Namespace gs::cnf and naming conventions

The configuration specific classes of GREENCONFIG have their own namespace gs::cnf which is a sub namespace of the GreenSocs namespace gs.

This framework uses the GREENCONTROL namespace gs::ctr.

A `using namespace ctr;` statement in the GREENCONFIG global import file imports the control namespace to the gs::cnf namespace (see greencontrol/gcnf/plugin/config_globals.h).

The most important classes of GREENCONFIG are available within the GreenSocs namespace gs as well (using statements). The classes are: `gs_param`, `gs_param_base` and `gs_param_array`.

For the correct namespace of the classes used in this document please refer to the doxygen generated API reference.

The classes of GREENCONFIG that are visible to the user have the prefix `GCnf`.

The GREENCONFIG service, and this documentation make use of some abbreviations:

- *GCnf* stands for *GreenConfig*.
- *cnf* or *config* stand for *configuration*.

SystemC delimiter The GREENCONFIG framework is developed using the dot (.) as the SystemC delimiter within names, which follows the standard. The file gc_globals.h defines SC_NAME_DELIMITER where this delimiter can be changed if the SystemC implementation e.g uses a slash (/). Although this is not OSCI standard it should work for most cases.

2.2 Files

These are the GreenConfig files that are of interest for the user. The project can be downloaded from the GreenControl project web site (as part of the package *greencontrol*).

Example files (IP, Tool,...) are located in the subdirectories of greencontrol/examples. A demonstration platform is in the directory greencontrol/examples/demonstraion_platform. See the GREENCONTROL User's Guide for a listing of the GREENCONTROL files.

Most important examples:

- greencontrol/examples/regression_tests :
  Different test IPs which use different parameters and a test tool which uses the GREENCONFIG `Tool_Api` to set and get parameters of the TestIP.
- greencontrol/examples/demonstration_platform :
  Demonstration platform (testbench file testbench_demonstration_platform.cpp and configuration files demoplatform<x>.cfg .
- greencontrol/examples/user_api_example :
  Example API which uses the gc_ports directly (file tool_api/tool_api.h).
- greencontrol/examples/gcnf_private_params :
  Example with private parameters and private APIs.
- greencontrol/examples/parameter_arrays :
  Example with Simple Parameter Arrays.
- greencontrol/examples/extended_parameter_arrays :
  Example with Extended Parameter Arrays.
- greencontrol/examples/callback_example :
  Example showing the parameter's callback functionality.
- greencontrol/gcnf/apis/commandLine_Tool :
  User API adapter (class `CommandLine_Tool`) which provides a command line inside the simulation terminal.
- greencontrol/gcnf/apis/commandLineConfigParserApi :
  User API adapter (class `CommandLineConfigParser`) which reads in parameters from the command line.

greencontrol	
config.h	Main user include file. Includes the config plugin, the standard GCnf API and the parameters.
config_api_command_line_parser.h	Includes the command line parser which handles command line arguments setting parameter values.
config_api_command_line_tool.h	Includes the tool which provides a command line during the simulation, allowing direct configuration and inspection during simulation runtime.
config_api_config_file_parser.h	Includes the configuration file parser.
config_api_lua_file_parser.h	Includes the LUA configuration file parser.

greencontrol/docs/GreenConfig	
users_guide/latex/GCnf_Users_Guide.pdf	This User's Guide.
tutorial/latex/GreenConfig_Tutorial.pdf	Tutorial of use.
api_ref/index.html	Link page to API Reference.

greencontrol/gcnf/apis/scmlApi	
scml_property.h	Objects which can be used as parameters in the user code. All objects of one module use together one instance of the `Scml_Api`.

greencontrol/gcnf/apis/toolApi	
tool_gcnf_api.h	User API adapter class `Tool_GCnf_Api` for initial and runtime configuration which uses the `GCnf_Api`.

2.3 Quick Start

To use GREENCONFIG at least the following include is needed:

```
1 #include "greencontrol/config.h"
```

Recommended: The sc_main should initialize the framework before instantiating any user modules:

```
1 int sc_main(int argc, char *argv[]) {
    /// GreenControl Core instance
    gs::ctr::GC_Core        core;
    // GreenConfig Plugin
5   gs::cnf::ConfigPlugin configPlugin;

    // User modules etc.

    sc_start();
10  return EXIT_SUCCESS;
  }
```

Alternative If you don't care on which classes are included, alternatively this short variant may be used, which includes and instantiates some more parts of GREENCONTROL and its sub-projects (e.g. GREENCONFIG, GREENAV):

```
// Standard GreenControl with config and analysis
#include "greencontrol/all.h"

int sc_main(int argc, char *argv[]) {
  // Standard GreenControl instantiations (Core, GreenConfig, GreenAV)
  GS_INIT_STANDARD_GREENCONTROL;

  // User modules etc.

  sc_start();
  return EXIT_SUCCESS;
}
```

- The include greencontrol/all.h includes many useful classes of several projects. Additional linking may be needed (e.g. link to lua). This may change independently of this documentation, see the code for the currently included files.
- The macro GS_INIT_STANDARD_GREENCONTROL has to be used in the main function (or equal) and instantiates several standard elements. E.g. these are the Core, the GREENCONFIG and GREENAV plugins. This may change independently of this documentation, see the code (file greencontrol/core/gc_globals.h) for the current configuration.

Short (experimental): As another alternative you can rely on the automatic creation of the GREENCONTROL Core and the Configuration Plugin with the default database (see section 2.5.2). In this case the sc_main needs not to initialize the framework:

```
int sc_main(int argc, char *argv[]) {

  // User modules etc.

  sc_start();
  return EXIT_SUCCESS;
}
```

2.4 Processing flow GreenControl and GreenConfig

In this section, an example flow shows what the framework does during construction time and preparation for simulation – if the recommended initialization style is used. Automatic instantiation differs.

- The GC_Core is instantiated,
- The required plugins are instantiated (ConfigPlugin)
  - The gc_port searches for the global GC_Core instance and connects automatically.

- A default user config API is instantiated
  - The contained gc_port performs automatic binding to the Core.
- Parameters are instantiated:
  - A user API is instantiated (containing a gc_port)
    - The contained gc_port performs automatic binding to the Core.
- Parameters are added (with default value) to the Config Plugin.
- Other IPs are instantiated.
  - ...
- Instantiate Tool API
  - gc_port bound automatically (see above)
  - Command get (etc.) may be used
- Core (CallbackDispatcher) gets SystemC kernel callback start_of_simulation
  - Core calls start_initial_configuration in each API
    The APIs may perform further initial configuration.
  - After that Core calls end_initialize_mode in each API

2.5 Configuration Service CONFIG_SERVICE

2.5.1 Configuration Plugin ConfigPlugin

The plugin which provides the configuration service is the `ConfigPlugin`. In its `gc_port` the supported service is set to `CONFIG_SERVICE` and the boolean `is_plugin` is set to true.

The ConfigPlugin provides and uses the commands listed in the following section. Pointers to parameters are stored in a *parameter database* and observer addresses are stored in a separate *observer database*. Both databases have to implement interfaces (`param_db_if` and `observer_db_if`) and are used by the ConfigPlugin, see figure 2.1.

The framework stores the values of the parameters not in the database but in the parameter objects themselves. The parameter database holds 'only' pointers to the parameters. This makes access to parameters fast.

Compatibility

⚠ Deprecated

For compatibility (to the GREENCONFIG framework revision 0.2) a User API still is allowed to add new parameters by name and value, not by pointer. In this case the plugin creates a parameter object and adds that pointer to the database. All `gs_param` usages and API method calls keep the same for compatibility. Some of the calls are deprecated due to speed reasons. When deprecated methods are called, an SC_WARNING is reported (macro DEPRECATED_WARNING) which can be disabled by defining `#define NO_DEPRECATED_WARNINGS` in greencontrol/core/gc_globals.h or the makefile/compiler settings.

The GREENCONFIG standard parameter database is the `ConfigDatabase`. It has to be instantiated in the top-level (testbench) – if not created automatically – and is given to the ConfigPlugin constructor. The parameter database can be replaced by another one, e.g. an adapter to the scml database (registry, server). Note the following design rule:

Design rule

⚠ Database and ConfigPlugin deletion

Create the database *before* the plugin and ensure that the deletion happens in reverse order!

The standard observer database is the `ObserverDatabase` which stores addresses of observers that are registered to be called when a new parameter is created. The observer database is instantiated and bound in the plugin. It gets a pointer to the plugin to be able to bind the `plugin_if` port.

The ConfigPlugin includes and construction can be done as follows:

```
#include "greencontrol/config.h"
// in sc_main (or elsewhere):
gc::cnf::ConfigDatabase db("CnfDb");
gc::cnf::ConfigPlugin configPlugin(&db);
```

⚠ Warning: Do not new the ConfigDatabase directly inside the plugin construction. With some compilers this may lead to an SystemC runtime error because of a failure in the SystemC module stack.

Another way creating the standard configuration plugin is calling the static get function. Note that this news the config plugin one time as singleton (if not created by the user before), strictly speaking this is a memory leak if is not been deleted after simulation – in the correct order after all APIs and parameters but before the core. This call works together with a previously manually created plugin.

```
#include "greencontrol/config.h"
// in sc_main (or elsewhere):
gs::cnf::ConfigPlugin::get_instance();
```

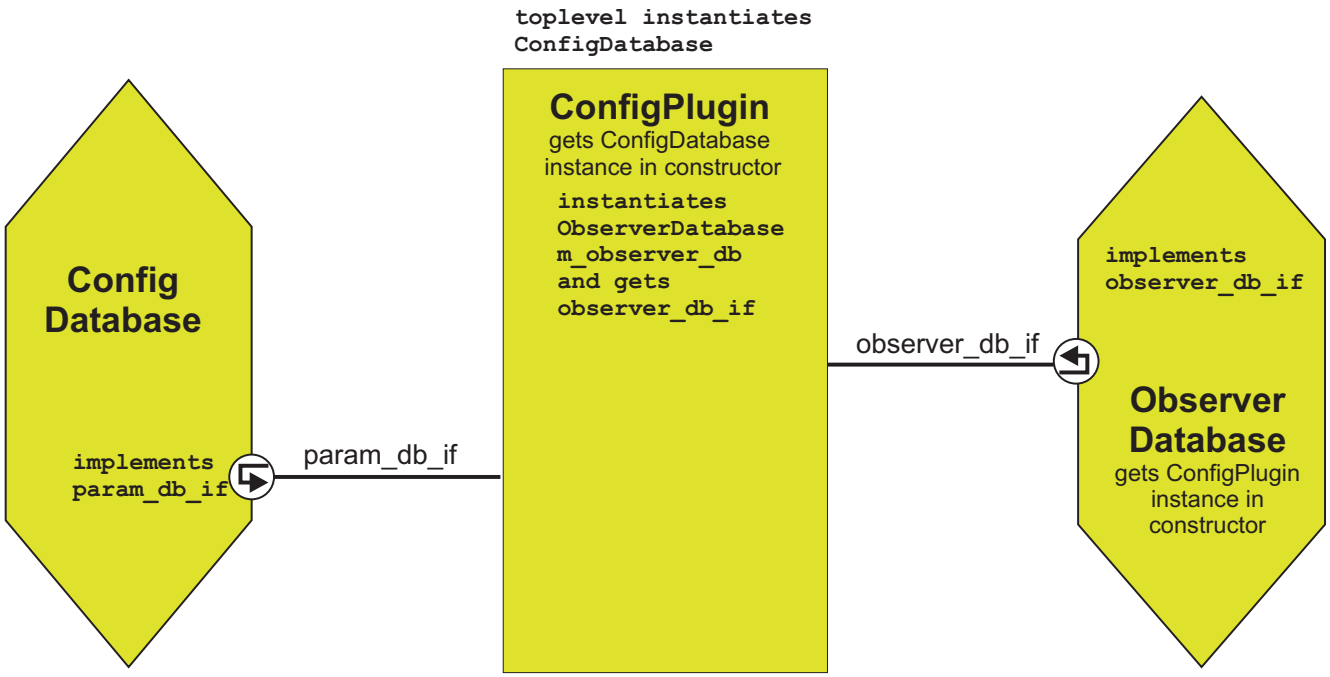

Figure 2.1: Connection of databases to the configuration plugin with interfaces.

2.5.2 Alternative automatic construction

⚠ Warning: **This is experimental.** Please report problems to christian.schroeder@greensocs.com!

Similar to the automatic GreenControl Core construction the configuration plugin can be automatically constructed - with its standard database.

On the first instantiation of any parameter or GCnf API the plugin will be created automatically.

- The plugin will be instantiated with the standard database class `gs::config::ConfigDatabase`.
- This only works if there is any instantiation of a parameter or GCnf API before the `sc_start` call!

2.5.3 Commands for the Configuration Service (internal)

In the configuration service (`CONFIG_SERVICE`) the command enumeration `ConfigCommand` is used. The plugin and the APIs have to know how to process each command. If a user wants to add a new command, it has to be added to the enumeration and the ConfigPlugin and the affected APIs have to be adapted. The plugin and the `GCnf_Api` throw a warning if the command is unknown.

All configuration command may be used during elaboration and runtime.

The following commands may be used by the Config APIs. The fields of the transaction are used for special commands on the shown way:

Direction: API → ConfigPlugin

Command	Phase	Field	Description
`CMD_ADD_PARAM`	Adds a parameter *and* sets its default value. This command may only be used by the parameter owning module. The default value has to be set with this command and *not* with a following CMD_SET_PARAM to allow other modules (e.g. tool, config file parser tools) to overwrite the default value. This command creates an explicit parameter or converts an implicit to an explicit by not overwriting the implicit set init value.		
	REQUEST	*AnyPointer*	Parameter pointer.
	REQUEST	alternative *Specifier* and *Value*	Name of a parameter that should be created by the plugin.
			Value of the parameter that should be created by the plugin.
	RESPONSE	*Error*	> 0 when adding fails.

`CMD_SET_INIT_VAL`	Sets the init value of a parameter. If the parameter does not yet exist, it is created as an implicit parameter. The init value of an implicit parameter has priority to the initial value of a new added parameter. This command overrides the default value which was/will be set during adding a parameter. This gives tools the opportunity to set initial values. See section 2.5.4.		
	REQUEST	*Specifier*	Parameter name.
	REQUEST	*Value*	Init value of the parameter.
	RESPONSE	*Error*	No error specified so long.

`CMD_LOCK_INIT_VAL`	Locks the init value of a parameter. Lock so that this parameter's init value cannot be overwritten by any subsequent `CMD_SET_INIT_VAL`. This allows to emulate a hierarchical precendence since a top-level module can prevent the childs from setting init values by locking the init value before creating the subsystem. Returns false (and does not lock) if parameter is already existing explicitely. Returns false (and does not lock) if no initial value is existing that can be locked.		
	REQUEST	*Specifier*	Parameter name.
	RESPONSE	*Error*	> 0 when parameter lock was not applied.

CMD_GET_VAL	Gets the string representation of the parameter value. May be used for implicit and explicit parameters.		
	REQUEST	*Specifier*	Parameter name.
	RESPONSE	*Value*	String representation of the parameter's value.
	RESPONSE	*Error*	> 0 when parameter does not exist implicit or explicit.

CMD_GET_PARAM	Gets a pointer to a parameter. This should not be done for a none existing parameter. (The error flag will be set in the transaction.)		
	REQUEST	*Specifier*	Parameter name.
	RESPONSE	*AnyPointer*	Pointer to the parameter.
	RESPONSE	*Error*	> 0 when getting fails.

CMD_REMOVE_PARAM	Removes a parameter from the plugin.		
	REQUEST	*AnyPointer*	Pointer to parameter.
	RESPONSE	*Error*	> 0 when removement not successful.

CMD_EXISTS_PARAM	Returns if the parameter exists. To return true the parameter may exist as implicit or explicit parameter.		
	REQUEST	*Specifier*	Parameter name.
	RESPONSE	*Error*	$= 0$: Parameter existing $= 1$: Parameter not existing

CMD_GET_ ↩ PARAM_LIST_VEC	Returns a vector of parameter names. Depends on flag mSpecifier: empty: Gets list of all existing parameters in the plugin. modulename (e.g. jpeg.encoder): Returns all parameters of the specified module (without its child modules). modulename.* (e.g. jpeg.*): Returns all parameters of the specified module including the parameters of its child modules.		
	REQUEST	*Specifier*	Empty or module name or <modulename>.*
	RESPONSE	*AnyPointer*	Vector of names (strings) of the requested parameters (all existing or all of that module or all of that module + childs). Receiver must delete the pointer!
	RESPONSE	*Error*	No error specified so long.

CMD_REGISTER_NEW_PARAM_OBSERVER	Registers an observer for new added or new set parameters.		
	REQUEST	*ID*:	Observer's address.
	RESPONSE	*Error*:	> 0 when registering failed.

CMD_UNREGISTER_PARAM_CALLBACK	Unregisters all parameter callbacks for the specified observer module. The callbacks were previously registered by the module directly at the parameters.		
	REQUEST	*AnyPointer*:	Pointer to the parameter whose callbacks should be unregistered.
	RESPONSE	*Error*:	No error specified so long.

CMD_PARAM_HAS_BEEN_ACCESSES	Returns if the parameter has ever been used.		
	REQUEST	*Specifier*:	Parameter name.
	RESPONSE	*Error*:	= 0 : used = 0 : not been used

Direction: ConfigPlugin → API

CMD_NOTIFY_NEW_PARAM_OBSERVER	Notifies an observer about new added (or without add first time set) parameters.		
	REQUEST	*AnyPointer*:	Pointer to the new parameter.
	REQUEST	alternative *Specifier* and *Value*	Name of the new parameter.
			Value of the new parameter.
	RESPONSE	*Error*:	No error specified so long.

2.5.4 Parameter types implicit, explicit

Think of a quite normal scenario: A module creates and sets a parameter (using its API). This creation and setting is done with the command CMD_ADD_PARAM (Attention: The API may only use CMD_ADD_PARAM, *not* CMD_SET_INIT_VAL). Afterwards an other API (e.g. a tool or a config file parser) changes the value of this parameter with CMD_SET_INIT_VAL. The parameter will be set to the init value set by the tool. ⇒ Everything works as expected.

Problem: The execution order of modules is random (respectively not predictable). Also the order of Core callbacks is random (see GREENCONTROL User's Guide). We want to keep this behavior so that all APIs can be treated in the same way. A tool API should not be treated differently from an other API. ⇒ That can lead to the case that a tool sets a parameter which is not yet exiting because the parameter owning module has not yet created the parameter.

Solution: The setting of a not (yet) existing parameter (with CMD_SET_INIT_VAL) is possible and results in an *implicit parameter*.

The API (e.g. a tool API) which wants to set a parameter which is possibly not yet existing may use CMD_SET_INIT_VAL to create an implicit parameter. An implicit parameter will be converted to an *explicit parameter* when the owner module (API) itself makes use of the CMD_ADD_PARAM command. Only the parameter is allowed to add/create an explicit parameter. This is obvious because during construction the parameter's API performs the add itself..

2.5.5 Initial value vs. default value

If an implicit parameter – which has an *initial value* (*init value*) – is converted with the add command to an explicit parameter, the existing init value of the implicit parameter is *not* overwritten by the *default value* submitted in the add command. So the default value is of less relevance than an init value. This makes sense because obviously a tool wants to change the initial value. To avoid problems with the setting order, the `setInitValue` API call (or the CMD_SET_INIT_VAL command) should be used only once. Likewise, as expected, a set init value command after a add command overwrites the default value. The user should avoid setting the init value of a parameter more than once, e.g. do not use a config file tool (see sec. 2.6.12) and a command line config parser tool (see sec. 2.6.13) to set the same parameter twice. Anyway the second call will overwrite the value of the first one. In the GreenConfig API (see section 2.6.3) the call `setInitValue(string &parname, string &value)` results in such an init value.

Type	Description
implicit	A parameter which was created only with the CMD_SET_INIT_VAL command (without leading CMD_ADD_PARAM).
explicit	A parameter which was added (by the existing parameter itself) with the CMD_ADD_PARAM command.

2.5.6 Unconsumed initial values

There is a way to find out if there are any parameters (and which ones) that are not (yet) consumed (unused) (`is_used`). A parameter is not used if it never has been explicit and the implicit paraneter (the initial value) had never been read. Thus it is possible to find out (for analysis tool like e.g. functional coverage tools) after the simulation ended which parameters never had been used, e.g. because they had some typos in their names in the config file. There is a way to bypass this mechanism to access the initial value without marking the parameter as used (to e.g. show the value in the tool).

2.5.7 Lock initial value to emulate hierarchical precedence

To emulate a OVM-like hierarchical precedence for initial values an initial value can be locked after it has been set. Such a locked iniitial value cannot be overwritten by any subsystem (when created after the lock) so the higher level module or tool has the control. In the GreenConfig API (see section 2.6.3) the call `lockInitValue(string &parname)` applies such a lock.

2.5.8 Order of execution

The `gc_port` connects immediately to the Core so it may immediately submit its parameters to the Config Plugin. If the desired commands are processed in the constructor of the API (e.g. parameter constructors result in an add command) the user can effect the order of the commands arriving at the Config Plugin by instantiating the IPs in the testbench in a special order.

If there is an API/module which configures parameters at its construction time the instantiation of the configuration APIs effects the priority or order of the parameter settings: Before instantiation of the user module which owns a parameter the last change to this parameter value will be the actual one.

2.6 API adapters

*API adapter*s (User APIs) allow to translate between the GREENCONFIG framework and other configuration APIs. The `GCnf_Api` is the native User API. Other User API adapters for the configuration service may use the `GCnf_Api` or directly the `gc_port` to access the service.

The framework may be used immediately after construction of the Core and the needed Plugins. During instantiation of the `gc_port` it is bound automatically to the Core.

The following sections present some of the existing User APIs.

2.6.1 Concept configuration APIs

There is one native configuration GREENCONFIG API which should be used within the user code to access functionality of the plugin that goes beyond the parameter's abilities (see section 2.6.7). This API is described in detail in section 2.6.3.

The simulation should contain only one instance of this API which is created by the configuration plugin during instantiation. The user should access this API by calling the static function `getApiInstance` (see section 2.6.3.1). The returned pointer is either the one API instance or a private API responsible for the calling module. This returned pointer is of the type `gs::cnf::cnf_api_if` which is mainly an interface which is implemented by the two APIs.

Legacy Info

GCnf_Api coding rule changed

You should not instantiate a `GCnf_Api` object any longer (deprecated) but get the (only) existing API by calling the `getApiInstance`-function. Together with this change the type of the used API has been changes to `gs::cnf::cnf_api_if` – which has the same functions as the config API itself.

Legacy Info
GCnf_Api instances
The concept did change from one API per module to one API per simulation. Main advantages are that the user needs not to be aware if the simulation is already running and that the function returning the API pointer is able to choose the standard or a private API.

Another concept, used e.g. for the SCML API, is having one API instance for each module. See figure 2.2 for an example which includes both concepts. One SCML API for each module is created, but only one GCnf API for the simulation.

See section 2.6.3.1 for details on the GreenConfig API's static get instance function.

2.6.2 GreenConfig API Interface (cnf_api_if)

To achieve independency of the underlying API type (normal 2.6.3 or private 2.6.6) both APIs implement the class `cnf_api_if` which is mainly an interface with virtual functions to be implemented by the APIs and some additional functions implemented in this class. A pointer of type `cnf_api_if*` is returned by the `getApiInstance` function so the user acts on this one.

Most API functions are virtual and are implemented by the two APIs. Some special functions are independent of the API type and are templated so they cannot be virtual.

Please consider the first two functions are convenience functions but not necessarily the recommended way getting a parameter value, because they are of bad performance:

- `template<class T> const bool getValue(const std::string &parname, T &value)`
  returns a parameter's value independent of the implicit or explicit status. The string value is converted to the user-chosen template type by using the `gs_param` template specialization. This function does only work for types of `gs_param<T>`, not for any kind of `gs_param_base`, e.g. parameter arrays.
  This function calls the virtual `std::string getValue(...)` function to get the value from the underlying API. Please consider this function is of bad performance!
- `template<class T> const T getValue(const std::string &parname)`
  is a similar function which copies the value instead of writing it to the value parameter. Please consider this function is of bad performance!
- `template<class T> gs_param<T>* get_gs_param(const std::string &parname)`
  is a convenience function which casts the parameter base object to the user-specified type of `gs_param`.

2.6.3 GreenConfig API (GCnf_Api)

The GreenConfig API (class `GCnf_Api`) is the native GreenControl configuration API which provides the functionality of the framework to parameter objects, other APIs or the user directly. The GreenConfig parameters (`gs_param`, see section 2.6.7) and all other provided configuration APIs make use of this API.

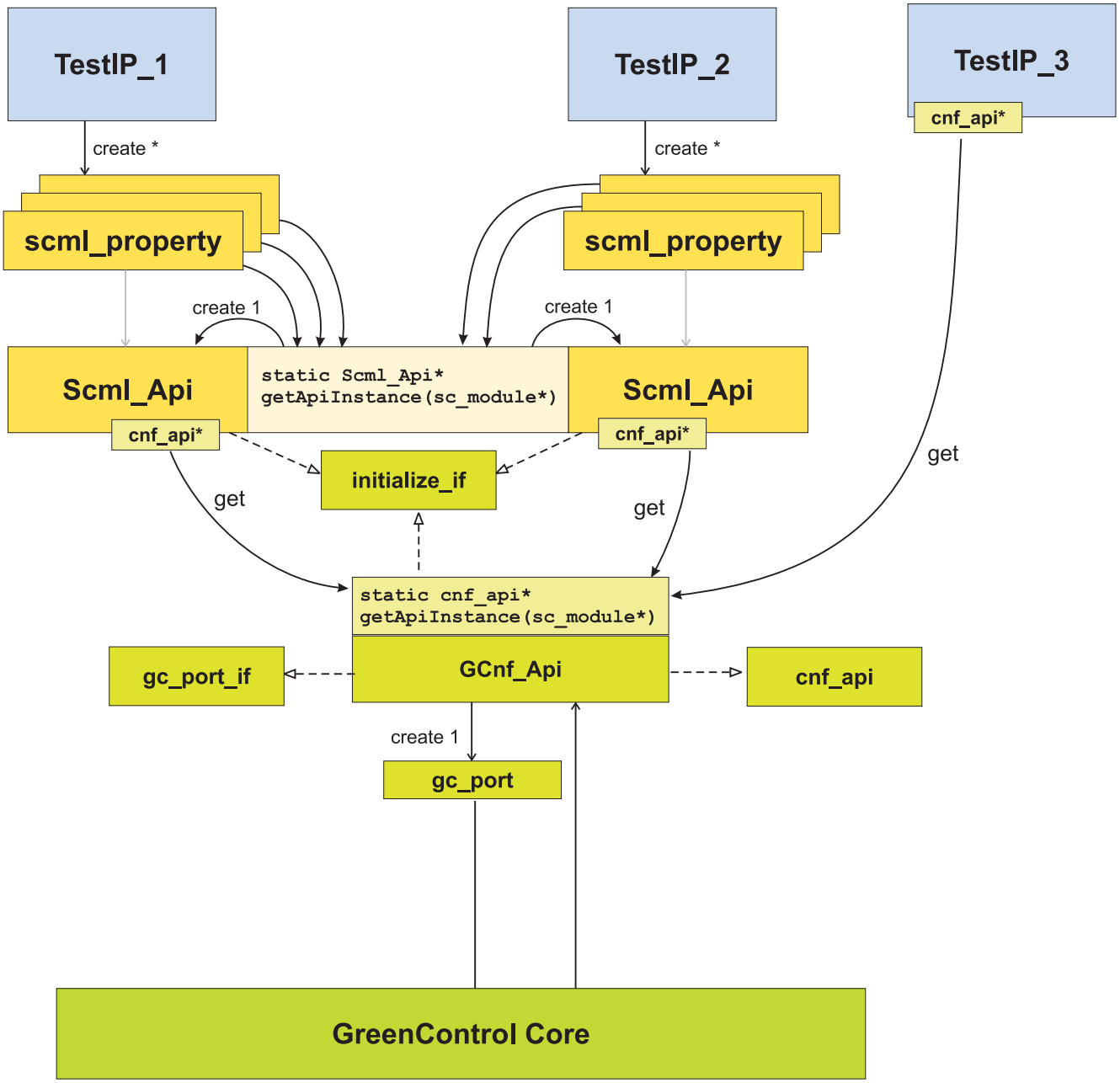

Figure 2.2: Different concepts according API instances for the SCML API and the GCnf API.

In the example user_api_example a simple API is implemented which does *not* use this API but uses the `gc_port`s directly.

The user should use the static get function instead (see 2.6.3.1) to get an API instance. The default `GCnf_Api` object is instantiated by the Config Plugin and should not be instantiated by the user directly.

For a full list of all functions provided by the GREENCONFIG API refer to the (doxygen) API reference (see the classes `gs::cnf::GCnf_Api_t<...>` and `gs::cnf::cnf_api_if`).

2.6.3.1 Static get API instance function

The GREENCONFIG API provides a `getApiInstance` method (cf. section 2.6.1) which returns a pointer to an API instance. This function should be called by a user to get the API instance which is responsible for the module:

```
gs::cnf::cnf_api_if* gs::cnf::GCnf_Api::getApiInstance(sc_module*);
```

The returned type `gs::cnf::cnf_api_if` is an interface being implemented by the standard configuration API and also by a special API providing private parameters (see section 2.6.6). The user module automatically gets the correct (standard or private) API, accordingly all actions performed on the API pointer automatically respect whether the actions are on private parameters or not.

Only the top-level testbench (or other special tool class) is allowed to call with the parameter being set to NULL – which will return the default API instance

2.6.3.2 Set initial values

The function `bool setInitValue(parname, string value)` sets an initial value (see section 2.5.5) to – possibly not yet existing, hence implicit – parameters.

2.6.3.3 Lock initial values

The function `bool lockInitValue(parname)` locks an initial value (see section 2.5.7) so that this parameter's init value cannot be overwritten by any subsequent setInitValue call. This allows to emulate a hierarchical precendence since a top-level module can prevent the childs from setting init values by locking the init value before creating the subsystem.

Returns false (and does not lock) if parameter is already existing explicitely or if no initial value is existing that can be locked.

2.6.3.4 Status is_used

With the API function `bool is_used(const std::string &parname)` you can check if a parameter has ever been used. It returns `true` if the parameter has ever been explicit or the implicit (initial) value has ever been read. This allows for analysis (functional coverage) tools to check after the simulation for e.g. typos in the config file. There is an option for such tools to bypass the status update when calling the API function `getValue` by setting `not_impact_is_used_status` to `true`.

2.6.3.5 Parameter access

Access to any parameter in the system can be done over the API – as long as the parameter is not private. The standard way accessing a parameter is calling `getPar(param_name)` on the API, e.g.

```
gs_param_base* par = mApi->getPar("any_module.param1");
// do some inefficient stuff, see gs_param section
// check if par != NULL
par->setString("10");
std::string val = par->getString();
int ival;
par->getValue(ival);
```

If you know the type of the parameter, use e.g.

```
gs_param<int> *par = mApi->get_gs_param<int>("any_module.param1");
// do more efficient stuff, see gs_param section
// check if par != NULL
(*par) = 10;
```

The functions `getType()` or `getTypeString()` can be used to get the type, see section 2.6.7.

The function `existsParam` checks if a parameter is existing explicit or implicit. To check for an explicit parameter, use e.g. `if (api->getPar(name)!= NULL)`.

Each time `api->getPar(name)` is used, it should be checked to be not NULL. If models configure each other it should not be relied on the existence of each other to make the models portable.

There are several functions to get multiple parameters as a vector:

- `getParamList(<optional string argument>)` returns a vector of parameter names (strings). Depending on its argument it returns all or a choice of parameters, see the Doxygen documentation for details.
- `getParams(<optional string argument>)` returns a vector of parameter (base) pointers.

2.6.3.6 New parameter callbacks

The macro `REGISTER_NEW_PARAM_CALLBACK` can be used to get called for new added parameters.

```
class MyIP_Class
: public sc_core::sc_module {
public:
  // some code [...]
  // Example code to register callback function
  void main_action() {
    // some code, parameters etc...
    my_GCnf_Api.REGISTER_NEW_PARAM_CALLBACK(MyIP_Class, config_callback);
  }
  // Callback function for new added params.
  void config_callback(const std::string parname, const std::string value)
    ;
  }
```

```
};
```

2.6.3.7 Parameter callbacks and event notifies

This API provides the function `unregisterAllCallbacks(void* observer)` which uses the GREENCONTROL command CMD_UNREGISTER_PARAM_CALLBACK. This unregisters the specified observer module from any existing parameter stored in the database. This function is not used by the framework during destruction of an observer but may be used by an user if needed.

⚠ Deprecated: Instead of registering callbacks or getting events for parameter changes at the GREENCONFIG API please use directly the parameters to register there, see section 2.6.7.1.

2.6.3.8 Config API initialize-mode

The GREENCONFIG API may be used during initialize-mode as well as during runtime-mode (see GREENCONTROL User's Guide).

2.6.4 How to make use of the GCnf Api in an API adapter

This section is about how to develop a new API for some special use by internally using the existing standard GCnf API. For an example see the Tool GCnf API in directory 📁 greencontrol/gcnf/apis/toolApi/ .

- The newly developed API file should be placed in an own subdirectory. GREENCONFIG APIs are places inside the configuration API directory, e.g. 📁 greencontrol/gcnf/apis/my_api
- The new API has to include the GCnf Api:

```
#include "greencontrol/config.h"
```

- The new API has to inherit `sc_object` and `initialize_if`.
- The new API should *not* be an `sc_module`.
- The new API should get a pointer to the `GCnf_Api` by calling `getApiInstance` only once during instantiation and should store the pointer.

```
My_Api()
  : sc_object(sc_gen_unique_name("__my_api__"))
{
  m_gcnf_api = gs::cnf::GCnf_Api::getApiInstance(
                    gs::cnf::get_parent_sc_module(this));
}
private:

/// GCnf_Api which is used by this API
gs::cnf::cnf_api_if* m_gcnf_api;
```

2.6.5 How to develop an API adapter directly using the gc_port

For an example see the Tool API within the GREENCONTROL example
greencontrol/examples/user_api_example .

- The newly developed API file should be placed in an own subdirectory. GREENCONFIG APIs are places inside the configuration API directory, e.g. greencontrol/gcnf/api/my_api
- The API can be placed into the namespace gs::cnf.
- The API has to include the configuration include file (for class `gs_param_base`):

```
#include "greencontrol/config.h"
```

- This examples presumes having used the control and config namespaces:

```
using namespace gs::ctr;
using namespace gs::cnf;
```

- The API has to inherit `sc_object`, `gs::cnf::initialize_if` and `gs::ctr::gc_port_if`.

```
class My_Api
: public sc_object, // Better try without!
  public initialize_if,
  public gc_port_if
```

- The API should *not* be an `sc_module`.
- The interface `gc_port_if` has to be implemented (method `transport`).
- The API has to instantiate the `gc_port` to communicate: `gc_port m_gc_port;`. The service, which has to be specified in its constructor, has to be set to CONFIG_SERVICE (for configuration APIs). The unique API name has to be set and the bool value (if this is a plugin) has to be set to false. this has to be given to the port.

```
My_Api()
  : sc_object(sc_gen_unique_name("__tool_api__")),
    m_gc_port(CONFIG_SERVICE, MY_API_NAME, false, this)
{
}
```

- The interface `initialize_if` *can* be be implemented and can be used to identify the initialize-mode and to do initial configuration if this is not possible at construction.

2.6.6 GreenConfig private API (GCnf_private_Api)

The GREENCONFIG private API (class `GCnf_private_Api`) is a special API that allows private parameters. The private API implements the interface `gs::cnf::cnf_api_if` and can be used exactly as the normal API.

A module which should hide its parameters and the ones of its childs can instantiate a private API. This API is registered at the static `getApiInstance` function. If one of the parameters or child modules call the static `getApiInstance` function it will return the private API. The private API has an internal parameter database similar to the configuration service plugin.

The user can give parameters to the private API that should *not* be private by giving their local names to the private API constructor.

Hierarchies of private APIs are possible, see figure 2.3. More details are shown in figure 2.4.

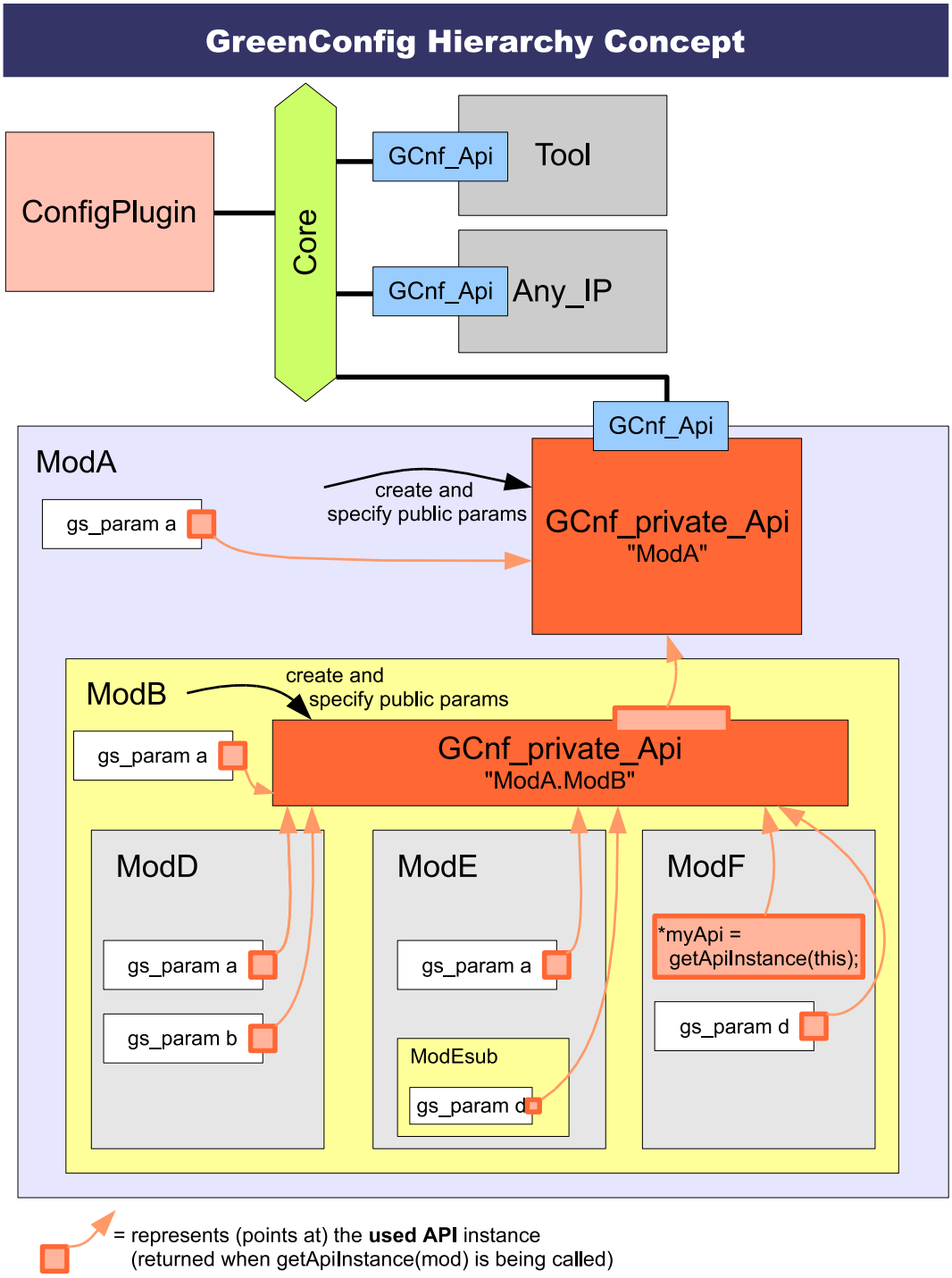

Figure 2.3: Concept database hierarchy with private APIs.

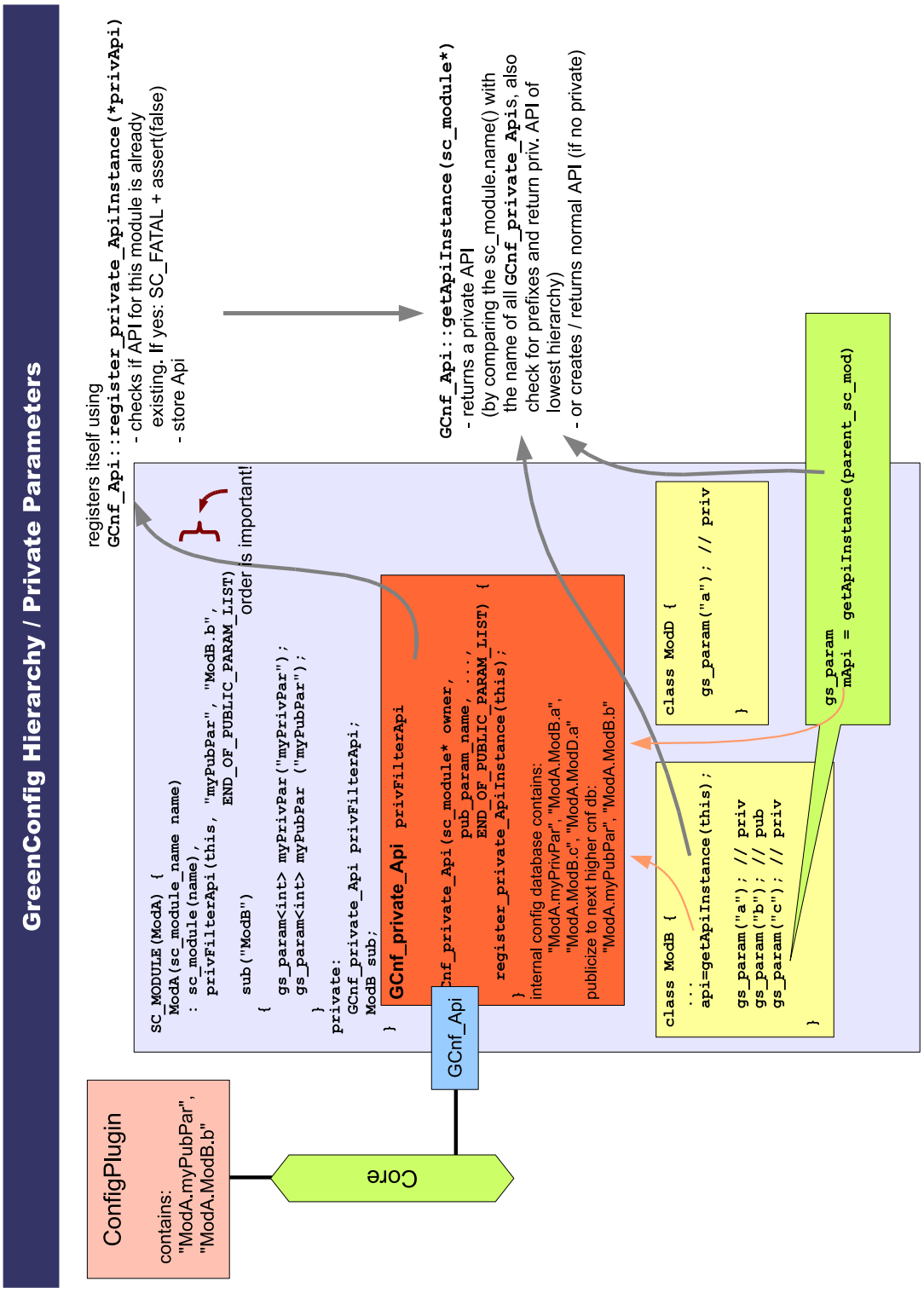

Figure 2.4: Database hierarchy with private APIs.

Usage The example below shows a module (ModA) which creates a private API (privFilterApi). That API makes all parameters of ModA and parameters of its submodules private. Additionally calling `getApiInstance(mod)` returns this private API as long as mod is a pointer to the owning module or one of its child modules. Exclusions are the two parameters myPubPar and ModB.b.

The *instantiation order* is important: First instantiate the private API, afterward instantiate any parameters and submodules.

The private API object must exist at least as long as any parameter or child module which is accessing this API exists.

When constructing the private with specifying public parameters, the constructor with variable arguments is the default one. Use `END_OF_PUBLIC_PARAM_LIST` to finalize the parameter name list. The parameter names are *local parameter names*! The default constructor is:

```
GCnf_private_Api(sc_module* owner_module, const char* pub_par ...);
```

Alternatively the constructor without public parameters can be used:

```
GCnf_private_Api(sc_module* owner_module);
```

At the end the constructor getting the parameter name list as a vector can be used instead of the on with variable arguments:

```
GCnf_private_Api(sc_module* owner_module,
                 std::vector<std::string> pub_params);
```

The module pointer must point to a valid module. Private top level parameters are not supported.

```
class ModA
  : public sc_module {
  [...]
  ModA(sc_module_name name)
  : sc_module(name),
    privFilterApi(this, "myPubPar", "ModB.b", END_OF_PUBLIC_PARAM_LIST),
    sub("ModB"),
    int_param("int_param", 45),
  {
    gs_param<int> myPrivPar("myPrivPar");
    gs_param<int> myPubPar ("myPubPar");
  }
private:
  gs::cnf::GCnf_private_Api privFilterApi;
  ModB sub;
  gs::gs_param<int>   int_param;
}
```

Submodules need not to be aware of the private API and can instantiate parameters and get their API pointer normally. The private API will be used automatically.

```
class ModB
  : public sc_module
  {
  public:
    SC_HAS_PROCESS(ModB);
    ModB(sc_module_name name)
    : sc_module(name),
      submod("AnotherSubModule"),
      b("b", 1111),
    {
      mApi = gs::cnf::GCnf_Api::getApiInstance(this);
    }
  protected:
    AnotherModule submod;
    gs::gs_param<int> b;
    gs::cnf::cnf_api_if *mApi;
  };
```

2.6.6.1 Implementation

A private API calls the static `getApiInstance` function during construction before registering itself. Accordingly the private API gets a pointer to the next API upwards responsible for the owning module (*parent API*). This may be the (one) normal top-level API but this also may be another private API. This allows to build a hierarchy of private APIs.

API calls according the specified public parameters are forwarded to the parent API, all other calls are directed to an internal private config database which is like the config service plugin. The communication does not use transactions but function calls to make it less vulnerable. The private API checks for each call whether to forward it to the parent API or whether to process it locally. There are also some calls that are mixed (e.g. `getList()` returns a combination of the parent and the private list).

2.6.7 GreenConfig parameters: gs_param

The main interface a normal user gets in touch with when using the GreenConfig framework are the GreenConfig parameters, called *gs_param*.

Usage The favored parameters that should be used when a new configurable IP is developed are the GreenConfig parameters in the class `gs_param`.

The `gs_param` class is – additional to the configuration namespace gs::cnf – also available within the GreenSocs namespace gs.

These parameters support the whole functionality of the GreenConfig API (`GCnf_Api`) and can be used just like the data types they represent.

New data types can be added by creating template specializations (`gs_param<T>`).
The supported data types are:

- int, unsigned int, bool, double, float, string, unsigned long long, long long, unsigned char, unsigned short, char, signed char, sc_int_base, sc_int, sc_uint_base, sc_uint, sc_signed, sc_bigint, sc_unsigned, sc_biguint, sc_logic
  (with specialized classes including all operators)
- sc_attribute<T> with T= all other supported types
- sc_bit are deprecated, use bool instead.
- sc_time can deserialize the following strings to sc_time:
  - "<double value>" → Convert to seconds
    (reverse way of the `sc_time::to_double()` function)
    Example: `my_param.setString("0.000000025"); //sets value to 25 ns`
  - "<uint64><sc_time_unit>" (reverse way of `sc_time::to_string()` function)
    Example: `my_param.setString("25 ns"); //sets value to 25 ns`
    Example: `my_param.setString("25 SC_NS"); //sets value to 25 ns`
- std::vector<std::string> serializes a vector to e.g.

```
{"This","is the","simplest","example"}
{"example","with","quotes \"here\" inside and comma, here","and single quote 'here"}
```

  Syntax to set the string representation of such a parameter:

  Example C++ code:

```
vecLocal.setString(std::string("{\"example\", \"with\", \"quotes \\\" here\\\" inside and comma, here\", \"and single quote 'here\" }"));
```

  Example config file (not-lua):

```
IPVec.vec5 "{ "with", "quotes in \\"this\\"." }"
## we need this double \\ syntax to be the same as in code (\" needs to be in the resulting string being set)
```

- All data types which support the stream operators may be used as template specialization for `gs_param`.

Before the parameters can be used the developer has to include the parameter class (see file greencontrol/gcnf/apis/gs_param/gs_param_class.h) by including greencontrol/config.h .

Parameters are derived from a base class `gs_param_base` which can be used each time the type does not matter or is not known, e.g. as a function's parameter or return value. The `gs_param_base` class is – additional to the configuration namespace gs::cnf – also available within the GreenSocs namespace gs.

Usage:

```
  gs::gs_param<int> int_param("my_par_name", "104");
// or
  gs::gs_param<int> int_param("my_par_name", 104);
```

Usage as member variables:

```
class MyIP
: public sc_module
{
public:

  SC_HAS_PROCESS(MyIP);
  MyIP(sc_module_name name)
    : sc_module(name),
      int_param ("int_param"),
      uint_param ("uint_param", 1000) // with default
  {   }

  /// Example parameters.
  gs::gs_param<int>           int_param;
  gs::gs_param<unsigned int>  uint_param;
};
```

Some different *constructors* may be used to instantiate a `gs_param`:

- Constructor with the name of the parameter,
- Constructor with the name and the *default value* (value type or string representation).

The constructor without name is private and cannot be used because it results in an unnamed parameter.

For integer values hexadecimal notation may be used.

The *operators* `=`, `+`, `-`, `/`, `*`, `+=`, `-=`, `/=`, `*=`, `%=`, `^=`, `&=`, `|=`, `<<=`, `>>=`, `--` (prefix, postfix), `++` (prefix, postfix) are (as possible) available for the supported data types, see itemization above.

Also the conversion operator `gs_param<T>::operator const T& ()` is implemented.

For purposes of GREENAV (Analysis and Visibility service plugin) there are the operators `+`, `-`, `/`, `*` for the parameter base class `gs_param_base`. The operators (called 'convenient operators') return a tripple which is used by the Calculator in GREENAV.

In most cases the parameter instance may be used like the data type it represents. To get the value, simply use the parameter variable, and to set the value use the = operator. If that does not work (e.g. for not explicit supported data types) use the data type methods `setValue(T)` and `const T& getValue()`.

Operators `==` may be used to compare parameters of same type to each other or with objects of their type. Examples:

```
gs::gs_param<bool> myBoolParam1, myBoolParam1; // initialization skipped
bool result = (myBoolParam1 == myBoolParam2);
bool result2 = (myBoolParam1 == result);
bool result3 = (result2 == myBoolParam2);
```

Parameter string representation

To *get* and *set* with string representation of the values the methods `setString(const string&)` and `const string getString()` can be used. See the doxygen documentation for rules for each parameter type.

The parameter string set functions support the environment variable substitution which is described in section 2.6.12 (cf. section 2.6.7.2).

Parameter attributes

Parameters may have a set of attributes which give more information about their usage within the model.

Parameter attributes (`gs::cnf::param_attribute`) (don't mistake them for sc_attributes) are simple `unsigned int`s, represented with an enumeration: `gs::cnf::param_attributes::param_attribute_enum`. The possible types are: (TODO) config, runtime_config, elab_config, read_only, analysis, temp, output, internal.

Parameters provide the following functions to give access to the attributes:

- `bool add_param_attribute(const param_attribute attr)`
  adds a parameter attribute.
- `bool has_param_attribute(const param_attribute attr)const`
  returns if a special parameter attribute is set.
- `const std::set<param_attribute> get_param_attributes()const`
  returns a set containing all parameter attributes that are set for this param.
- `bool remove_param_attribute(param_attribute attr)`
  removes a parameter attribute.
- `void remove_all_param_attributes()`
  removes all parameter attributes of this parameter.

The following listing shows how to use the parameter attributes:

```
gs::gs_param<int> p("param");
p.add_param_attribute(gs::cnf::param_attributes::config);       // add
p.add_param_attribute(gs::cnf::param_attributes::output);       // add
if (p.has_param_attribute(gs::cnf::param_attributes::config))   // check
  /* do something */
set<gs::cnf::param_attribute> s = p.get_param_attributes();  // get all
p.remove_param_attribute(gs::cnf::param_attributes::output); // remove
p.remove_all_param_attributes(); // remove all
```

Parameters not registered at the database

There may be situations where it is reasonable to create parameters that are not registered at the database (regardless whether this is the normal or a private one). The advantages of parameters can be used without the need of making them accessible via the API and plugin (database).

⚠ Be careful when using these parameters, normally standard parameters should be used!

Two special constructors are available to create such parameters (simplified view). The name and value need to be provided as strings.

```
gs_param(string &name, string &value, gs_param_array* parent_array,
   bool force_top_level_name, bool register_at_db);
gs_param(string &name, string &value, gs_param_array& parent_array,
   bool force_top_level_name, bool register_at_db);
```

Parameters as array members

Another set of constructors can be used to create parameters as extended array members. See section 2.6.8.2 for details.

Parameter base class gs_param_base

Some functions in the configuration framework return the base class of the specialized parameter class if type independency is needed (e.g. callbacks).

A pointer/reference to a parameter base object (`gs_param_base`) can be useful in different ways:

- Give it directly to many available functions that get parameter bases.
- Get the string representation of the parameter's value: `string getString()`
- Set the the parameter's value using its string representation: `bool setString(val_str)`
- Get the value already converted to the right type. The awaited type `T` must be known:
  `bool getValue<class T>(T &value)` (returning the success of the conversion) or
  `T getValue<class T>()` (one more memory allocation because of copying the return value).
- Get the parameter pointer already converted to the right type:
  `gs_param<T>* get_gs_param<class T>()`
- Convert to the *unknown* template type:
  The base class `gs_param_base` has two functions to get the original type of the object which allows the user to cast the base object to the correct parameter object: `getType()` returns an enumeration `Param_type` which may be used in an switch case statement. This function returns reasonable values for the supported data types. When using user specialized data types the function `getTypeString()` may be used. There a string describing the type is returned.

For more details about how to use parameters see the GREENCONFIG Tutorial.

Parameter Base: get void* value The virtual function `virtual void* get_value_pointer()` can be used to cast the value to a base class of the value, e.g. cast an `sc_int<length>` to `sc_int_base`.

A use case are SystemC data types where it is not possible to cast a `gs_param_base` to the corresponding type, e.g. `gs_param<sc_int<10> >`. In this case call `get_value_pointer()` on the base parameter and cast the result e.g. to `sc_int_base`.

Example:

```
void callback(gs::gs_param_base &par) {
  switch (par.getType) {
    case gs::cnf::PARTYPE_SC_SIGNED:
    case gc::cnf::PARTYPE_SC_BIGINT:
      void* pv = par.get_value_pointer();
      sc_signed* p = static_cast<sc_signed*>(pv);
      // Do something with p->length() and p->to_uint64() etc.
  }
}
```

Parameter names Users creating parameters should use *parameter names* matching the SystemC naming rules. A parameter name should not include SystemC name delimiters (e.g. dots for OSCI implementation).

⚠ But delimiters are not forbidden, a user who knows what he is doing is allowed to use delimiters!

There is the possibility to *choose the name of a parameter fully free* of the hierarchical position of the owning module. Even within a module it is possible to choose a arbitrary name (matching the naming rules, including delimiters) and prevent the parameter from attaching the name prefix of the owning module. In this case the sc_object is still a child of the creating module, only the parameter's name may be in a different hierarchy position. To do this use the constructor parameter `force_top_level_name` which is an optional parameter for various constructors, see the (doxygen) API reference.

⚠ This feature may confuse users and should be used carefully!

Top-level parameters Parameters are allowed to be instantiated at top-level (outside the SystemC hierarchy), e.g. in sc_main. Here the same parameter naming rules are applied as for not top-level parameters.

⚠ Be careful not to cause name collisions with existing SystemC modules and their parameters when using hierarchical parameter names with delimiters in them.

Parameter name collisions Since each parameter name is unique in the database, it is an error to create more than one parameter with complete identical names. This could happen when setting top-level parameter names manually. During creation of the doubled parameters there will be an sc_warning. Initial values are applied to all parameters (matching the given name) that are tried to add to the database now, even if they are already existing and adding fails. It is strongly recommended not to use such not added parameters but resolve the naming collision.

2.6.7.1 Callbacks

This section is about how the user and APIs can be notified about parameter changes and other actions on parameters using function callbacks (sc_event-based notifies are deprecated).

Each parameter handles an own list of callbacks and an event notification. These notifies can be received by any object. E.g. a tool can be notified on all changes of any parameter.

Event (deprecated) The parameter object is able to return a reference to an *sc_event* which is notified each time the parameter value changes: `sc_event& getUpdateEvent()`.

Callback A callback function can be registered and unregistered with the parameter. This is the preferred way due to performance reasons. Callbacks also work before simulation runtime.

Callback types There are different callback types (enum `callback_type`):

`pre_read`	Calls before a parameter value is been read. Value can be updated here.
`post_read`	Calls while a parameter is read. Value shall not be updated within this callback[1]
`reject_write`	Calls before the parameter value will be written to – allows the callee to reject the write.
`pre_write`	Calls before the parameter value will be definitely written to (not being rejected).
`post_write`	Calls after the parameter value write is done.
`create_param`	Calls when a new (implicit or explicit) parameter is created (to be registered with the GCnf_API, not a parmeter).
`destroy_param`	Calls when the parameter is to be destroyed.

⚠ Note : There is no post callback following a pre callback if the action failed.

Callbacks can be registered for any method with the signatures

```
gs::cnf::callback_return_type config_callback(gs::gs_param_base& ↩
    changed_param, gs::cnf::callback_type reason)
```

and register it at the parameter as callback function for this parameter. One callback function may be registered at several parameters and several callback functions may be registered at one parameter.

Inside callback functions it is forbidden to use wait statements.

Callback functions can be registered at a parameter with:

```
GC_REGISTER_TYPED_PARAM_CALLBACK(&my_param, callback_type, class_name, ↩
    callback_method_name)
```

To enable a module being able to register callbacks, use the macro `GC_HAS_CALLBACKS()` in the module's body. This macro creates a member variable which stores the created callback adapters to be able

[1] Technically this is not really a post_read because the get function not yet returned to the user. Even the actual read from the containing value within the parameter might not have been done yet if it is done directly within the return statement.

to unregister them when destructing the module. In addition to this the macro creates a help function which is needed when registering a callback.

The user module needs to call the macro `GC_UNREGISTER_CALLBACKS()` in its destructor. With this macro all callbacks that were registered by this module will be unregistered. A shared pointer is used to hold the adapter instance to ensure that it is destroyed when all interested instances (parameter and observer module) are destroyed or unregistered.

Optional: If the user takes care to unregister all callbacks himself this callback needs not to be used - but it may be used in addition. If there is no reason to leave it out, use it!

```
class MyIP2 {
  GC_HAS_CALLBACKS();
  MyIP2() { ... }
  ~MyIP2() {
    GC_UNREGISTER_CALLBACKS();
  }
  ...
}
```

The following sniped is an example where IP1 has parameters which are observed by IP2 and IP3 (see greencontrol/examples/callback_example).

```
void MyIP2::main_action() {
  // Register post_write-callback for parameter int_param in module IP1
  GC_REGISTER_TYPED_PARAM_CALLBACK(m_Api->getPar("IP1.int_param"), gs::cnf
      ::post_write, MyIP2, cnf_callback);
  // Register pre_read-callback for parameter str_param in module IP1 with
       keeping the callback adapter pointer
  boost::shared_ptr<gs::cnf::ParamCallbAdapt_b> cbAdapt;
  cbAdapt = GC_REGISTER_TYPED_PARAM_CALLBACK(m_Api->getPar("IP1.str_param"
      ), gs::cnf::pre_read, MyIP2, cnf_callback);
}

// Callback function with default signature.
gs::cnf::callback_return_type MyIP2::cnf_callback(gs::gs_param_base&
    changed_param, gs::cnf::callback_type reason) {
  gs::cnf::callback_return_type cb_return = gs::cnf::return_nothing;
  switch (reason) {
    case gs::cnf::pre_read:
      // Here you could update the parameters value to whatever the
         reading object should see
      break;
    case gs::cnf::reject_write:
      // Here you could reject the write if it does not match some
         requirements, e.g. parameter dependencies
      if (reject) cb_return = gs::cnf::return_value_change_rejected;
      break;
    case gs::cnf::pre_write:
      // Here you could log the old value
      std::cout << "   callback function: (old value: "<<changed_param.
         getString()<<")" << std::endl;
```

```
      break;
    case gs::cnf::post_write:
      // Here you could log the new value
      std::cout << "   callback function: (new value: "<<changed_param.
          getString()<<")" << std::endl;
      break;
    case gs::cnf::destroy_param:
      // Here you should remove your local reference/pointer to this
          parameter to prevent from using it later (seg faults!)
      break;
    default:
      // nothing else possible
      // Note: create_param is called to a different method with different
           signature
      break;
  }
  return cb_return;
}
```

As can be seen in the example the register call returns a boost shared pointer to a parameter callback adapter (`boost::shared_ptr<gs::cnf::ParamCallbAdapt_b>`). This adapter pointer *may* be kept by the use module if it needs to unregister the callback *before* destruction of the module.

The callback function gets a reference to the base class of the parameter. The user can get the value directly only via a string (calling `getString()`). The base class can be casted to the specialized `gs_param` object using the `getType()` or `getTypeString()` functions.

The callback function must check the `is_destructing()` function on the parameter to check whether this is a callback to inform about the destruction of the parameter!

Callback order The callback order is the registering order (since GREENCONTROL release 3.0.0).

Unregister callbacks A callback that was registered at a parameter can be unregistered in different ways. The default way is item one. You need not to use the other ones!

1. *Default*:
   `GC_UNREGISTER_CALLBACK(boost::shared_ptr<gs::cnf::ParamCallbAdapt_b> callb)`
   Unregister a callback whose callback adapter pointer (shared pointer) is available in the user module.

2. Use the GREENCONFIG API call `my_Api.unregisterAllCallbacks(void* observer)`
   Unregister all callbacks the observer module (whose this pointer must be specified) has registered at any parameter in the system. This call uses the config plugin to call the unregister function of each existing parameter in the system.

3. `myparam.unregisterParamCallbacks(void* observer)`
   Unregister all callbacks the observer module has registered at this (myparam) parameter. This function is used by the config plugin when performing a the API call of item two and may be called directly by the user, too.

⚠ Attention : The callback function should take care when setting the parameter's value because that could result in a new callback of the same function which means an endless loop.

⚠ Attention : During a callback it is safe to unregister the current callback adapter. But unregistering further callbacks of that parameter can cause a segmentation fault because the parameter's callbacks are just called. If the parameter is destructing (destroy_param callback or `is_destructing()` returns true) it is fase to remove any callback of that parameter (and of course of any other parameter, too) because here a (slower) safe callback function is used.

Background destruction mechanism for callbacks The issue why the destruction mechanism with the macro `GC_UNREGISTER_CALLBACKS(mGCnfApi)` is needed for observer modules which register parameter callbacks is: 1. An observer module registers a callback at a parameter. 2. The observer module gets destroyed. 3. The parameter still has the callback registered: on changes or even on destruction of the parameter it tries to call the observer - which is already destructed 4. ⇒ segmentation fault

So the observer module must assure that each callback it has registered at any parameter is unregistered when the module gets destroyed!

The way chosen:

- Each observer module holds a list of its parameter callback adapters. This list is initialized by the macro `GC_HAS_CALLBACKS()` which must be used in the class body of the module that registers callbacks.
- When a new callback is registered at any parameter this callback adapter is added to the module's list.
- During destruction of the module these adapters are deleted. This is done by the macro `GC_UNREGISTER_CALLBACKS()` which must be added to the module's destructor.
- During their destruction the adapters inform their parent parameter (they have a pointer to it) to remove themselves from the parameter's callback list.

Another possible way:

- During callback registering the created parameter callback adapter stores the pointer to the observing module.
- When destructing the module, a macro could send the this pointer of the module to the ConfigPlugin via the API.
- The ConfigPlugin calls ¡code¿unregisterCallbacks(pointer)¡/code¿ on each known parameter.
- Each parameter searches for callback adapters which have the observer pointer set to that pointer and delete that adapters and callbacks.

2.6.7.2 Implementation

This section is about the implementation of the GREENCONFIG parameters and how to add template specializations.

The base class **gs_param_base** is not templated. It has member variables for the parameter name and the parent name which is used in the constructor (when no name is set) and for debug. The GreenConfig API variable is also a member of this base class. It is generated with the pointer of the next child object which is an sc_module. The member variables are protected to allow access by derived classes. The constructor instantiates the API instance using the static function `GCnf_Api::getApiInstance(mod)` (see section 2.6.3.1) and sets the parameter (and parent) name. The value-independent observer function is in this base class, too.

The template class **gs_param_t** inherits `gs_param_base` and extends it for some functionality which is dependent on the value type (operators, set and get value methods). All methods which need a conversion between value type and string representation make use of the virtual methods `serialize` and `getValue`.

The template class **gs_param** inherits `gs_param_t`. The template specializations should implement the operators (for the specialized data type) and must implement the conversion methods `serialize` and `getValue` to achieve a specialized conversion for the data type.

If no explicit specialization for the desired data type exists, there is the default `gs_param<T>` implementation. There the operators except the = operator are not available. Conversion is done per default with string streams.

How to make a new parameter type

If a specialization for a user data type (or an existing but not yet supported data type) is needed, the class `gs_param<T>` can be specialized for the needed data type T. The functions needed to be specialized are:

- type information functions,
- `serialize`,
- `static_deserialize`.

The macro `GS_PARAM_HEAD` creates some using statements, many constructors, the destructor and the non-static deserialize function which calls the user-implemented static one (for details see greencontrol/gcnf/apis/gs_param/gs_param.h). You also may (but need not) use operator macros.

It is recommended to use a class body similar to the existing specializations:

```
template<>
class gs_param<my_data_type>
: public gs_param_t<my_data_type>
{
  /// Typedef for the value.
  typedef my_data_type val_type;

public:
  GS_PARAM_HEAD;

  //  operators
  GC_SPECIALISATIONS_ARITHMETIC_OPERATORS;
  GC_SPECIALISATIONS_BINARY_OPERATORS;
  GC_SPECIALISATIONS_INCREMENT_OPERATORS;
```

```
15   GC_SPECIALISATIONS_DECREMENT_OPERATORS;

     /// Overloads gs_param_base::getTypeString
     const std::string getTypeString() {
       return string("my_data_type");
20   }
     /// Overloads gs_param_base::getType needs not to be overloaded
     //  because enum does not contain my type anyway
     //const std::string getType() {
     //  return PARTYPE_NOT_AVAILABLE;
25   //}

     /// Overloads gs_param_t<T>::serialize
     std::string serialize(const val_type &val) {
       [...]
30   }

     /// Static convertion function called by virtual deserialize and others
        (e.g. GCnf_API)
     inline static bool static_deserialize(val_type &target_val, const std::
        string& str) {
       [...]
35   }
};
```

In some cases the provided macros do not suffice, e.g. addtional constructors and operators are needed, even deriving from the base class `gs_param_t<Type>` sometimes is not possible. Just make sure the new parameter derives from `gs_param_base`. To follow the rules according string deserialization with environment variable substitution, make sure to call `envvar_subst(str, m_par_name)` within each deserialize call (compare file gs_param_t.h).

Rules for the coding of the functions:

- `getTypeString()`, `getType()`
  - The return value of `getTypeString()` may be any string that is unique within all specializations.
  - The function `getType()` returns a member of the enumeration `Param_type` (see greencontrol/gcnf/plugin/gcnf_datatypes.h) but needs not to be overloaded.
- `serialize()` (conversion value type → string):
  - May not make use of `m_par_name` because it is called inside constructor!
- `static_deserialize()` (conversion string → value type):
  - Set `target_val` to the default value if `str` is empty (`==""`).
  - If the conversion fails, do not change the value and return false.
  - If the conversion succeeded, return true.

- When overloading or adding value change (value read etc.) functions make sure to call the appropriate functions causing the callbacks and handling their return values (e.g. reject_write).

Enable debug outputs by defining `GS_PARAM_VERBOSE` for parameter outputs and `GS_PARAM_ARRAY_VERBOSE` for array outputs.

Destruction of parameters Because the database stores pointers to parameters the database / plugin need to know when the parameters are destroyed to be able to remove them.

- The destruction flag in the parameter base (class member variable `bool destruction_flag`) is set to true before destruction (in the destruct function `destruct_gs_param()` of the `gcnf_param class` which is called by the destructor of the `gs_param` object).
- The parameter calls makeCallbacks to inform all observers about the destruction. The observer callback functions must check the destruction flag by calling `bool is_destructing()` on the parameter reference it gets.
- `removePar(par)` in the `GCnf_Api` is called by the parameter destruct function after setting the destruction flag.
- The `GCnf_Api` uses the command CMD_REMOVE_PAR to send the destruction information to the plugin.
- The plugin calls the method `removeParam(gcnf_param_base *par)` on the database (`param_db_if` and `ConfigDatabase`).

Notes:

- Parameters (`gs_param_base` and `gs_param`) are prepared to be constructed by the plugin. In that case the parameter does not register itself at the plugin.

2.6.8 Parameter Arrays

GREENCONFIG provides two different types of arrays:

- Parameter arrays with *unnamed* members each of the *same data type*: *Simple Parameter Array*.
- Parameter arrays with *named* members of *different data types*: *Extended Parameter Array*.

Summary GREENCONFIG supports two different kinds of arrays: Simple and Extended Parameter Arrays.

Both array types are arrays of `gs_param`s whose members individually exist (as an object as well as in the database).

Here is a short summary of the differences of the two array types. For more details see the sections 2.6.8.1 and 2.6.8.2.

- The Simple Parameter Array's members are all of the same data type whereas Extended Parameter Array's members can be of different data types.
- Simple array members are unnamed and can be accesses by their position whereas extended array members are named and can be accessed by name.
- Simple arrays are of variable size, even configuration files are able to set an initial size and set as many members as it likes. Extended arrays are of fixed size and each member has to be constructed and named by their owner modules. Config files only can set existing members (when setting not existing members they will be ignored).
- Simple arrays have access functions like vectors (`[]operator`, `resize(int)`, `at(int)`) and have parameter functions (`getString()`, `setString("{member1 member2 member3}")`, etc.).
- Extended array members can be accessed using `find("memberName")` and nested arrays can be accesses using `getMemberArray("memberArrayName")`.
- Extended arrays have a callback behavior switch (see section 2.6.8.2) for details.

2.6.8.1 Simple Parameter Array

These parameter arrays are parameters of the type `gs_param<T*>` (see file greencontrol/gcnf/apis/gs_param/gs_param_array.hpp) which is a `gs_param` template specialization for T*.

The Simple Parameter Array contains members each of the same type `gs_param<T>`. They can be accessed by position (vector-like functions `my_array.at(pos)` and the operator `my_array[pos]`. The member's parameter name is <parent_array_name>.<pos>, e.g. 'my_module.my_sub_module.my_integer_array.0'.

Each member is stored as an individual parameter in the database. (This is needed to allow init values to be set e.g. by a config file with the style IP1.my_array.0 1000.)

The *array size* is set to one of the following sizes (with growing priority):

- The *default array size* 0 (zero).
  Can be modified within the scope of an object file by using the define DEFAULT_ARRAY_SIZE before including gs_param.h.
- *Highest implicit member number* '<array_par_name>.member_number' (in database): overwrites default array size. These members need not to be continuously (the user may set members .0 .1 .2 .100 .101 .2000 which will result in $size = 2001$).
- The *constructor array size* specified during construction (overwrites default array size as well as implicit members (except the .init_size).
- *Implicit parameter* '<array_par_name>.init_size' in plugin database (overwrites default and constructor array size).

The array is sized during initialization (construction). The members get their values either out of the default vector which was overgiven to the constructor or the members get their emtpy (default) value (if

not set by vector or by init value in the db) or the members get their init value out of the db (which has priority to all other values).

If the config file notation is used where each member is set in an own line (.0, .1 etc.) the config file tool sets the array size automatically if not given by the user with .init_size or by the constructor parameter. The user may set the init size manually, e.g. IP1.my_array.init_size 3.

If the user defines the init_size in the configuration file this has priority even higher than the size constructor parameter. If the user wants the array being sized to the number of members in the configuration file he either sets the init_size in the configuration file or he does not specify an array size in the constructor.

The array can be initialized with a vector of default values in the constructor.

- Advantage: A user module can simply instantiate a Simple Parameter Array and read out as many members as the config file has set.
- Disadvantage: Unnamed members and same member parameter types: less flexible compared to the Extended Parameter Arrays.

Different ways *accessing array members*:

- The `operator[]` is unsafe and allows access only to existing members (undefined behavior for others).
- Safe `at(position)` 'operator': If a member is accessed that is not existing it will be created. All members in between the last and the new highes member will be created with default values.

Usage How to use Simple Parameter Arrays:

Instances:

```
class IP1 {
  [...]
  gs::gs_param<int*>      my_int_array;
  gs::gs_param<float*>    my_float_array;
  gs::gs_param<string*>   my_string_array;
  gs::gs_param<unsigned int*> my_uint_array;
}
```

Constructor:

```
IP1()
 : my_int_array("my_int_array", 3),  // with size=3
   my_float_array("my_float_array"), // with default size
   my_string_array("my_string_array"),
   my_uint_array("my_uint_array", 3) // with size=3
 { ... }

 void any_method() {
   std::vector<int> intvec;
   for( int i = 0; i < 20; i++ ) intvec.push_back( 110 + i );
   // constructor with default values in a vector
   gs::gs_param<int*> my_new_arr("my_new_arr", intvec);
 }
}
```

Access:

```
any_function() {
   // set members individually
   my_int_array[0] = 100;
   my_int_array[1] = 101;
   my_int_array[2] = 102;
   // resize and set the new member
   my_int_array.resize(4);
   my_int_array[3] = 103;
   // or: resizes automatically and sets
   my_int_array.at(5) = 105;
   // set all members concurrently and resize automatically
   my_int_array.setString("{10 12 13 14 15}");
 }
```

Iterate:

```
for (int i = 0; i < arr.size(); i++) {
  cout << "member #" <<i<< "=" << arr[i].getString() <<endl;
}
```

Figure 2.5: Example for iterating through a Simple Parameter Array. Simple Parameter Arrays need no iterators because they can be accessed with the position.

Config file Usage How to use a configuration file to set Simple Parameter Arrays:

```
IP1.my_uint_array.init_size 3  ## optional.
IP1.my_uint_array.0 1000
IP1.my_uint_array.1 2200
IP1.my_uint_array.2 3330
```

Sets the array size either to init_size (if given) or automatically to the highest member (here size=3). This syntax has priority to the {...} notation.

This setting of members individually is supported by the GREENCONFIG config file API (configfile_tool.h). This API does *not* support the setting of all members at once (e.g. '{10 22 33}')! Look at the lua parser for that.

The config file setting members needs to be parsed before instantiating the parameter object to ensure the correct resizing and member values during construction of the array,

---

Experimental

Experimental

Use lua parser instead!

```
IP1.my_int_array 12 13 14
IP1.my_float_array 12.1 13.5 0.14 .3
IP1.my_string_array "Hello""world!""string with spaces + "double quotes"."
```

If you may use this notation see at lua parser config file.

Implementation details

The database needs not to be aware of arrays. The init size is just another parameter (child of the array). To achieve this independency the array constructor must read out the init_size parameter value (and cannot use a special command like 'get array size') and the implicit member names (to extract their numbers) to resize the array.

A Simple Parameter Array object explores its init size by

1. creating a member parameter called init_size which is added to the plugin database. If there was already an implicit parameter of this name the new created parameter will be set to that value. So after having created the parameter the new array object immediately checks the value and resizes itself to the correct size.
2. receiving a list of all implicit existing member parameters to calculate the array size (if it is greater than the init size).

```
1       class gs_param<T*>
        : public gs_param_t<T*>
        {
```

- The internal `my_value` is not used to store the members but a vector of `gs_param` pointers: `std::vector<gs_param<T>*> internal_ArrVec;`
- The *construction* can be done without default values and size, then the array is initialized with the default size, or the constructor gets the size or a vector of default values which also sets the size.

```
1       explicit gs_param(const std::string &nam, const unsigned int
            default_size = DEFAULT_ARRAY_SIZE)
        // or
          explicit gs_param(const std::string &nam, const std::vector<T
            > *default_vec = NULL)
```

The constructor creates the member parameter objects and stores their pointers in the `internal_ArrVec`.

- The destructor deletes the `internal_ArrVec` and all contained members.
- The getString functions returns a string of the syntax {"10" "12" "13"}.
- Functions in addition to normal parameters' functions:
  - The value handling functions setValue and getValue are not implemented because the `my_value` field is not used! Instead there are the vector-like getter functions `at(position)` and `operator[](position)` to get a member where the value handling functions can be called.
  - The `size()` function returns the size
  - The `is_simple_array()` function returns if the parameter is an Simple Parameter Array.
  - The `resize()` function resizes the array and deletes or creates members.
  - The `operator=(vector)` allows to set all values of the vector to the array members (also resizes automatically).
  - *Regression tests* see example parameter_arrays.

Config File Details The `init_size` parameter has to be set by the config file parser because only if the `init_size` parameter has an init value *then* the array constructor constructs the correct number of members which *then* are automatically set to the init values which are stored in the db.

So the `init_size` must be set (by the parser or the user) when setting init values for the members individually.

ToDo
Callback behavior
Simple Parameter Arrays: Callback behavior for parent array objects: Switch on/off for each observer if to call back when members change.

2.6.8.2 Extended Parameter Array

The Extended Parameter Arrays class is not a template specialization of `gs_param` but an own class which derives from `gs_param_base`.

An Extended Parameter Array can hold multiple parameters of possibly different types and even can hold other Extended or Simple Parameter Arrays.

The gs_param_array class is – additional to the configuration namespace gs::cnf – also available within the GreenSocs namespace gs.

Internally the array members are stored in a map of type map<string parname, par_base_ptr>.

- Advantage: More flexible than Simple Parameter Arrays due to named members, hierarchical arrays and different parameter types.
- Disadvantage: A config file can only set the members which are created by the user module.

Functions:

- public:
  - getMember(string local_param_name) returns the member with the specified name
  - getMemberArray(string local_param_name) returns the member with the specified name if it is an extended array (otherwise NULL)
  - operator[] see getMember
  - Constructor(string arr_name, gs_param_array* parent_array = NULL)
    - add myself to plugin db
    - store parent pointer
    - generate name

    Also constructors getting parent array references are available.
  - Destructor
    - remove this from plugin db and from parent array (call parent_arr->removeMember(this_name))
  - getType() returns "ExtendedArray"
  - is_extended_array() returns if the parameter is an Extended Parameter Array.
  - getString() returns a string representation of the array (including newlines)
  - size() returns the number of members
  - Macro SET_CALLBACK_BEHAVIOR(behav) sets the behavior of the callback mechanism for the setting observer (see below)
- protected:
  - addMember(gs_param_base* member) adds a member, called during construction of a member which got a parent_array. (friend class gs_param_base)
  - removeMember(string param_name) removes a member, called by the member's destructor. (friend class gs_param_base)

Usage How to use Extended Parameter Arrays:

Parameters (`gs_param`) and other arrays are added as a member by giving the parent array as a constructor parameter to the new object.

Parameters may get a pointer or a reference to the parent array:
`gs_param<T>(string par_name, string value_str_repr, gs_param_array &parent_array)` or
`gs_param<T>(string par_name, string value_str_repr, gs_param_array *parent_array)` or
several other constructors (see doxygen API reference).

Simple Parameter Arrays also may get a pointer or a reference to the parent array:
`gs_param<T*>(string arr_name, unsigned int default_size, gs_param_array &parent_array)` or
`gs_param<T*>(string arr_name, unsigned int default_size, gs_param_array *parent_array)` or
several other constructors (see doxygen API reference).

Extended Parameter Arrays also may get a pointer or a reference to the parent array:
`gs_param_array(string array_name, gs_param_array_T &parent_array)` or
`gs_param_array(string array_name, gs_param_array_T *parent_array` or
several other constructors (see doxygen API reference).

```
class IP1 {
  gs_param_array myTopArr;

  IP1()
  : myTopArr("myTopArr")
  {
    gs::gs_param_arr *subArr0 = new gs::gs_param_array
                                          ("my2ndArray0", &myTopArr);
    gs::gs_param_arr *subArr1 = new gs::gs_param_array
                                          ("my2ndArray1", myTopArr);

    gs::gs_param<int*>   *arrpar = new gs::gs_param<int>
                                         ("myIntArr", &subArr0);
    gs::gs_param<int>    *par1 = new gs::gs_param<int>
                                         ("myIntPar1", 12, &subArr0);
    gs::gs_param<string> *par2 = new gs::gs_param<string>
                                       ("myStringPar", "Def.", &subArr0);

    gs::gs_param<int>    *otherIntPar  = new gs::gs_param<int>
                                        ("myOtherIntPar", 500, &subArr1);
    gs::gs_param<int>    *otherIntPar1 = new gs::gs_param<int>
                                         ("myIntPar1", 600, subArr1);

    gs::gs_param_array    *subArray = new gs::gs_param_array
                                         ("mySubArray", subArr1);
  }
}
```

Figure 2.6: Example of usage. Alternatively the array member pointers could be class members.

```
myTopArr {
  'my2ndArray0',
  'my2ndArray1'
}
my2ndArray0 {
  myIntPar0 = 10,
  myIntPar1 = 12,
  myStringPar = "Def."
}
my2ndArray1 {
  myOtherIntPar = 500,
  myIntPar1 = 600,
}
```

Figure 2.7: Resulting structure

Access:

```
any_function() {
  gs::gs_param_base *tmp = m_Api.getPar("Owner.myTopArr");
  gs::gs_param_array &topArr = *(dynamic_cast<gs::gs_param_array*>(
     tmp));
  gs::gs_param_array::iterator it;
  it = topArr.getMemberArray("my2ndArray0")->find("myIntPar1");
}
```

Figure 2.8: Access an Extended Parameter Array in any module in the simulation (not the owner module which can access directly)

Config file Usage:

```
IP1.myTopArr.my2ndArray0.myIntPar0 100
IP1.myTopArr.my2ndArray0.myIntPar1 111
IP1.myTopArr.my2ndArray0.myStringPar "Hello world!"
IP1.myTopArr.my2ndArray1.myIntPar1 50000
```

Or use the lua config file parser (luafile_tool.h).

Iterators

The `gs_param_array` class has an iterator class `gs_param_array::iterator`.

The iterators support the operators =, ==, !=, ++ (prefix), – (prefix), *

The iterator class has the functions

- `begin()` returns an iterator which points to the first member.
- `end()` returns an iterator which points behind the last member.
- `find(string local_name)` returns an iterator which points to the member with the local name.
- `operator*()` returns a `gs_param_base`-pointer to the member.

```
gs::gs_param_array::iterator it;
for (it = myArr.begin(); it != myArr.end(); ++it) {
  cout << (*it)->getName() << "="<< (*it)->getString()<< endl;
}
```

Figure 2.9: Example usage of the array iterator.

Implementation details

- The array class registers an observer for each member. When being called the array can check if the member is being destroyed and then call removeMember for that member.
- *Regression tests* see example extended_parameter_arrays.

Callback behavior

A module which has previously registered as an observer at the parameter array can change the callback behavior of the array.

Per default CALLBACK_MEMBER_CHANGES is enabled: All changes of member values are forwarded to the registered observers of the array.

The behavior can be changed to NOT_CALLBACK_MEMBER_CHANGES: Only changes on the parameter array itself (and not member changes) are called back to the observer, e.g. changes of the size.

The observer can use the `SET_CALLBACK_BEHAVIOR(behavior)` macro to change the behavior which is stored individually for each observer. Observers are identified with their this pointers.
`behavior = CallbackBehavior::CALLBACK_MEMBER_CHANGES` or
`behavior = CallbackBehavior::NOT_CALLBACK_MEMBER_CHANGES`.

For details how to use the macro see example greencontrol/examples/extended_parameter_arrays
.

subsectionCoWare scml properties

The class **`scml_property`** (see greencontrol/gcnf/apis/scmlApi) is designed to replace the original CoWare properties. This property class uses the `Scml_Api` which exists once per property containing user module. The usage of the GREENCONFIG scml properties is the same as the original one.

Unlike the gcnf parameters the scml parameters cannot parse the hexadecimal notation for integers.

2.6.9 Tool_GCnf_Api

The **`Tool_GCnf_Api`** class (see greencontrol/gcnf/apis/toolApi) provides an API to work on parameters of other modules. The API is specialized to be used by tools which work with parameters of foreign modules. This API is used by the Command Line Tool (see section 2.6.10).

The available methods are listed in the API reference document generated by doxygen.

Example Tool API

API using `gc_port`

The `Tool_Api` class is another example API adapter which can be found in the example greencontrol/examples/user_api_example and uses directly the `gc_port` to set, get and list parameters of other modules.

2.6.10 Command Line Tool

The Command Line Tool may be included and instantiated in the testbench (class `CommandLine_Tool`, include file greencontrol/config_api_command_line_tool.h). It provides a simple command line, during simulation runtime, from the terminal.

Some simple commands are available: *get*, *set*, *list*, *h* (help), *q* (quit):

```
h    : this help
q    : quit
set <param_name> <value>  : Set value <value> of parameter <param_name>
get <param_name>          : Get value of parameter <param_name>
list                      : List all parameters
list <modname>            : List all parameters of module <modname>
list <modname>.*          : List all parameters of module <modname> and
                            its children
```

The only special use case is the `#define TEST_TOOL_POLL_TIME`. This define sets the interval the command line SC_THREAD polls for new data. For untimed models use `SC_ZERO_TIME`, for timed models use a reasonable time, e.g. `sc_time(1, SC_MS)`.

You need to `#define ACTIVATE_COMMAND_THREAD` to enable the tool.

Do not worry if the command line (`toolThread (h=help)>`) is not shown or disappears in other simulation produced output, just type the command and press enter.

Implementation The SC_THREAD `start_thread` starts an independent POSIX thread (pthread) which provides a console to the user in which commands can be types, and actions initiated.

When a full line is entered (enter is pressed) the data is written into the tool member variable `mThreadData` and the member variable `mNewThreadDataAvailable` is set to true. In the tool the `SC_THREAD poll_thread_data` checks these variables and calls `process_input()` if new data is available. This is done to allow the pthread to act fully independently from the simulation. Synchronization is done through `mNewThreadDataAvailable`. The thread waits with getting new input until `mNewThreadDataAvailable` is false.

2.6.11 Command line options

This section is about how GreenControl helps the user to handle command line options (arguments).

All parsers that need to handle command line options (section 2.6.12, 2.6.13, 2.6.14) are prepared to use either the linux *getopt* call or the *boost program options*. The getopt call is unix dependent but needs no further include or linking. That's why it is the default. The boost program options are operation system independent (e.g. work on Windows as well) but need the boost_program_options library linked. The user can choose the mechanism by defining USE_GETOPT (e.g. use compiler switch -DUSE_GETOPT) - which is the default and needs not to be specified - or by defining USE_BOOST_PROG_OPT (e.g. use compiler switch -DUSE_BOOST_PROG_OPT).

All parsers parse the options they know and ignore all others. On this way it is possible to call all parsers one after the other with the same (complete) command line options given to the sc_main.

The parsers should use long options with the prefix gs_ and no short (one character) options by default.

Note

Short program options

There is a global switch which enables the short (one character) program options (if available): ENABLE_SHORT_COMMAND_LINE_OPTIONS.

2.6.12 Config File Parser (ConfigFile_Tool)

The **`ConfigFile_Tool`** (include file greencontrol/config_api_config_file_parser.h) configures parameters from one or more configuration file(s) during elaboration (initialize-mode). It uses the `GCnf_Api`.

Usage with method calls:

```
#include "greencontrol/config_api_config_file_parser.h"

int sc_main(int argc, char *argv[]) {

  [... other modules ...]

  ConfigFile_Tool configfileTool;
  configfileTool.config("file.cfg");
  configfileTool.config("other_file.cfg");
  [command line arguments, see below]

  sc_start();

  return EXIT_SUCCESS;
}
```

Usage with command line options: To submit the command line options to the ConfigFile Tool the method `parseCommandLine` has to be called with the command line arguments (see section 'Command line options/switches'). This can be done by inserting the following line at the above mentioned position `[command line arguments, see below]`:

```
configfileTool.parseCommandLine(argc, argv);
```

The command line option (get help with `--help`) is:

- `--gs_configfile <filename>`

This option may be used multiple to read more than one configuration file.

Configuration file format:

```
# comment
hierarchical.parameter.name value
# e.g.
mymodule.submodule.parameterString  valueString
jpeg.compression 50
```

String parameters with spaces are supported. Possible syntax is:

```
jpeg.name I am a string parameter.
jpeg.name2 "I am a string parameter, too."
jpeg.name3 "I am a string parameter with \"quotes\" inside"
```

To have a full description (but don't use, may be changed in future):

```
jpeg.name4 ignored "I am a string parameter, too." ignored
jpeg.name5 ignored \"I am a string parameter with \"quotes\" inside\" ignored
```

`jpeg.name5` results in the value `I am a string parameter with "quotes" inside\` .

Configuration macros Configuration macros ease the use of replicated values.

Use configuration macros in text configuration files like following example:

```
BUSWIDTH = 32
Top.master0.busWidth BUSWIDTH
Top.slave0.busWidth BUSWIDTH
```

Rules: Configuration macros ...

- only contain upper letters and underscores (_)
- get their value assigned by '=' (contrary to normal parameters)

String macros are allowed, e.g.:

```
MY_MACRO = "my macro string"
MyMod.SubMod.StringParam MY_MACRO
```

Environment variable substitution Substitution for environment variables use the Makefile format $(varname). To avoid the substitution, use $$. Example:

```
Top.welcome.msg Hello $(USER), welcome to the simulator.
Top.master.memdump "$(DATA_DIR)/dump.bin"
Top.example "This will not be substituted: $$(foo)"
```

Config file notes

- Characters can be set using `'a'`.
  ⚠ Note: The surrounding `'`s are important! Otherwise a '0'=48 differs not from 0.
- Separators between parameter name and value may be spaces or/and tabs.
- The same parameter is allowed to be set multiple times within one or several config file(s). The last value will be applied to the parameter.
- The config file parser uses the `setInitValue()` API function. Accordingly the parameter value is the same, independently of the execution order parsing the config file and instantiating the parameter. Better follow the next rule for all cases:
- For some special parameter types it is important to *parse the config file before instantiating the parameter*. This is the case if the parameter constructor will search for implicit member parameters (e.g. Simple Parameter Arrays, Bits of GREENDEV registers).

2.6.13 Command Line Argument Parser

The `CommandLineConfigParser` API (see include file greencontrol/config_api_command_line_parser.h) reads parameters values from the command line arguments and sets the values in the Config Plugin using the `GCnf_Api`.

Command line parsing can be done in two different ways:

- The constructor `CommandLineConfigParser(sc_module_name name, int argc, char *argv[])` gets the command line options and starts immediately the parameter configuration.
- The constructor `CommandLineConfigParser(sc_module_name name)` instantiates the parser but does no parsing. The parsing can be done later by calling `parse(int argc, char *argv[])throw(CommandLineException)`.

Command line usage:

```
program_name [-other switches or arguments] [--gs_param <param_name>=<↩
   param_value>]* [others]
```

Possible Options/Arguments:

```
--gs_param parname=value     multiple args possible
--gs_param "parname=value"
--gs_param parname="value"
```

```
Values with quotes: '--gs_param parname="this is a value string with \" quotes\""

--help
```

Example:

```
./myProgram.x --gs_param IP2.scmlIntParam=1000 --gs_param IP1.scmlStringParam=testValueString --gs_param IP3.stringPar="This is a string with \"quotes\" and spaces"
```

If *duplicate parameter names* are uses the last one will be the one which remains in the Config Plugin.

The *order of calling* e.g. the config file parser and the command line parser has an effect on the values of parameters if the same parameters are set in both APIs. If the config file parser is called first and afterwards the command line parser is called, the command line value will overwrite the config file value.

The *setting time* of the parameter values for the Config File Parser (see 2.6.12) and the Command Line Config Parser (see 2.6.13) is dependent on the call of the parse-constructor or the parse-method. Setting may be done at elaboration time (e.g. in the main method of the testbench) or may be done during runtime. For example different configuration files may be applied at different simulation steps.

2.6.14 Lua Parser

The lua parser is able to parser configuration files with lua syntax.

Please visit the website for information on the lua parser.

2.7 Communication and function call flow

Figure 2.10 shows different use cases and their communication and function call flow. The following itemization shows the different steps that can be seen in the figure.

- The first use case is the '*compatibility mode*':
  Two legacy objects (tool and module) using string based function calls of the GREENCONFIG API.
  - The legacy module calls the string based `addParam` function in the API.
  - The API creates a transaction with the command CMD_ADD_PARAM using the string fields Specifier and Value.
  - The Config Plugin instantiates a parameter instance and adds this to the database.
  - Now the legacy tool calls the string based `getParam` function.
  - The API translates this call to a parameter pointer based call `getPar` which creates a transaction with the command CMD_GET_PARAM using the AnyPointer field to get the parameter pointer.
  - The API calls `getString` on the parameter and returns this string value to the legacy tool.

- The same translation is done by the API when being called by the legacy tool to set the parameter string based.

- The second use case is '*normal mixed with compatibility mode*':
  A (normal) module owning a parameter and two tools (one normal and one legacy) accessing it.
  - The module instantiates a parameter instance which registers itself at the plugin using its GREENCONFIG API.
  - The API creates a transaction with the command CMD_ADD_PARAM using the pointer field AnyPointer and sends it to the plugin.
  - The plugin adds the pointer directly to the database.
  - The legacy tool now accesses the parameter using the string functions of the API (see first use case).
  - The normal tool uses the call `getPar` in the API to get a parameter base pointer.
  - The API uses a transaction with the command CMD_GET_PARAM using the AnyPointer field to get the parameter base pointer (exactly like the translated call of use case one did) and returns this to the tool.

- The third use case is '*normal*' one:
  A (normal) module owning a parameter and a (normal) tool accessing it.
  - The (normal) module instantiates a parameter instance which registers itself at the plugin by calling the pointer based `addPar` call of the parameter's API.
  - The API creates a transaction with the command CMD_ADD_PARAM using the pointer field AnyPointer and sends it to the plugin.
  - The plugin adds the pointer directly to the database.
  - The tool uses the call `getPar` in the API to get a parameter base pointer.
  - The API uses a CMD_GET_PARAM transaction containing the string name in the Specifier field.
  - The ConfigPlugin returns the parameter base pointer in the AnyPointer field.
  - The API returns the parameter base pointer (of type `gs_param_base`) to the tool.
  - The tool can either use the string representation of the parameter base or cast the returned parameter base to the correct `gs_param<data_type>` pointer.

2.8 Notes

- Parameter names may contain characters like variable names in C++ may (e.g. numbers, letters, underline character).

- Like XParam the GREENCONFIG framework is able to parse command line parameters. We support a similar extensibility: user defined data types can be supported. They only have to be representable as string. The User API which uses these types has to convert between string and data type.

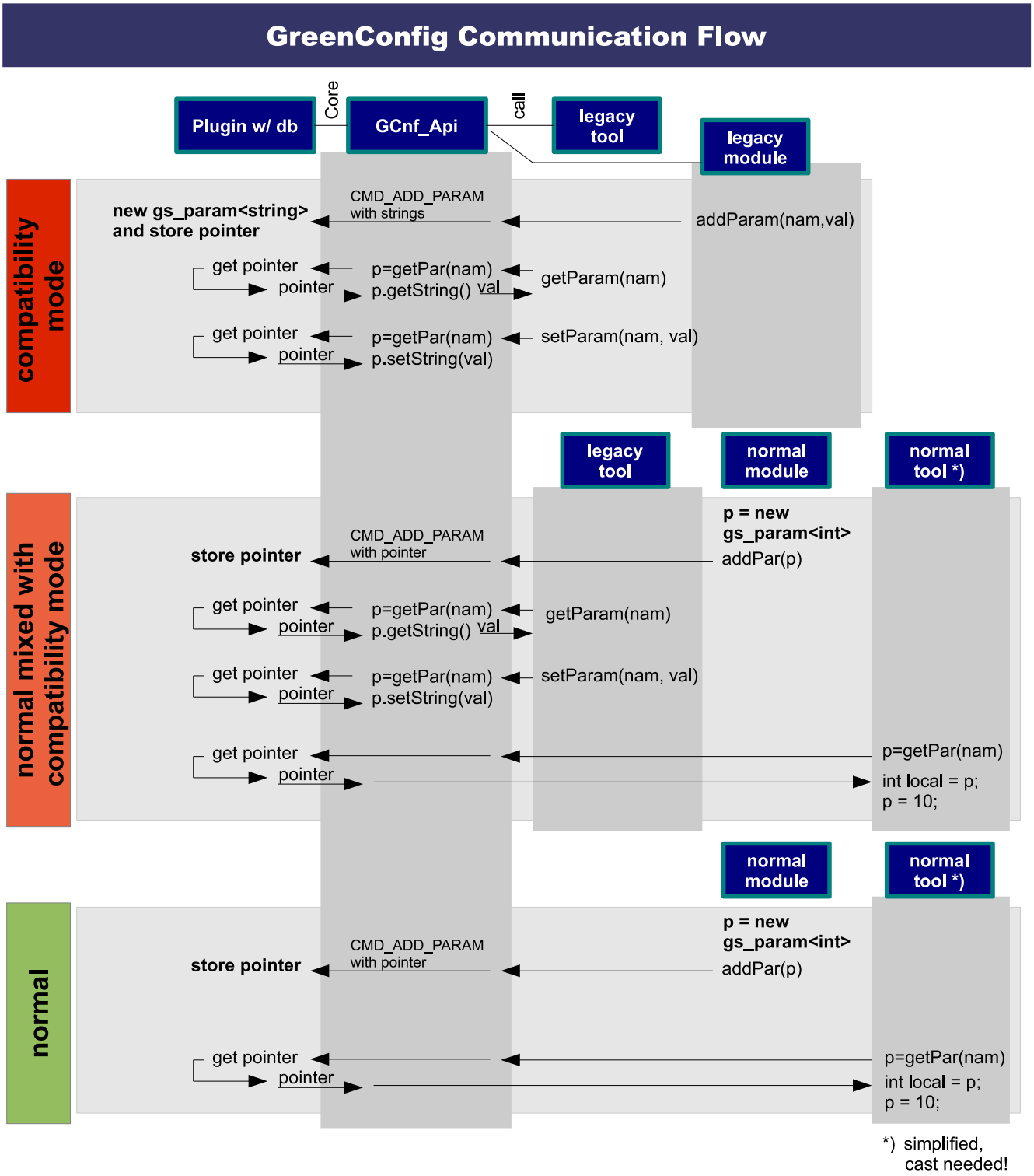

Figure 2.10: Three use cases with their GreenConfig communication and function call flow.

2.9 Implementation code

Visit the GreenSocs web page to get the newest revision of the GREENCONFIG framework:
http://drupal.greensocs.com/projects/GreenControl/GreenConfig

Appendix A

Appendix

A.1 Requirements for the GreenConfig Configuration Framework

A.1.1 Definitions

Before we start, we need some definitions:

A.1.1.1 Tool

We can consider two types of tools:

- *Internal* tools, i.e. a configuration/debug library linked to the model
- *External* tools may assist in verification of the model by providing increased visibility and control during simulation, and assisting the user in configuring the model in various ways before and during simulation.

A.1.1.2 Configurable Parameter

By the term configurable parameter we denote sc_module members that can be accessed (get/set value) by means of a config API.

A.1.1.3 Config API

A config API is an interface to a configuration framework with the goal, that

- the user can create and access (get/set) parameters
  - during construction/elaboration
  - at simulation runtime

Additional features may include

- tracking of parameter value changes (e.g., using event notification or callbacks)
- initialize parameters from a configuration file
- search for parameters in the model
- get a list of all parameters in the model
- dump parameters to a trace or config file
- add constraints / assertions to parameters
- synchronize parameters between SystemC model and external tool (push / pull)
- ...

Examples for different config APIs are:

- CoWare SCML
- ARM CASI
- Intel DRF
- GreenSocs Simple Configuration Framework

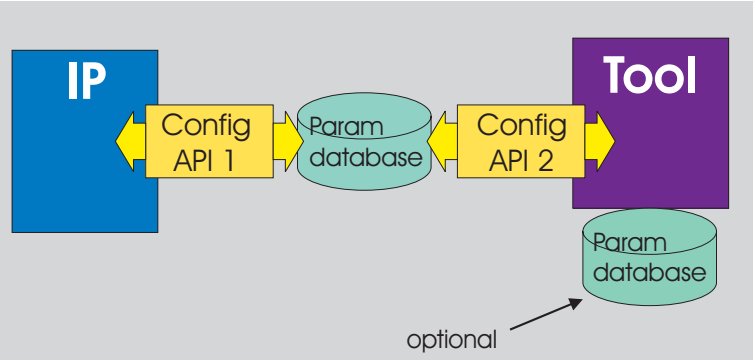

Figure A.1: Overview configuration in general.

A.1.2 Basic idea: the GreenConfig Core

The GREENCONFIG Core implements all functionality required to manage configurable parameters in SystemC models (implementation details follow below). It provides a generic GC-API which is used to built heterogeneous User APIs on top of it.

Hence it has to be well designed to support virtually all functions other APIs might ask for. **User API adaptors** translate between the GREENCONFIG framework and the vendor's interfaces / APIs.

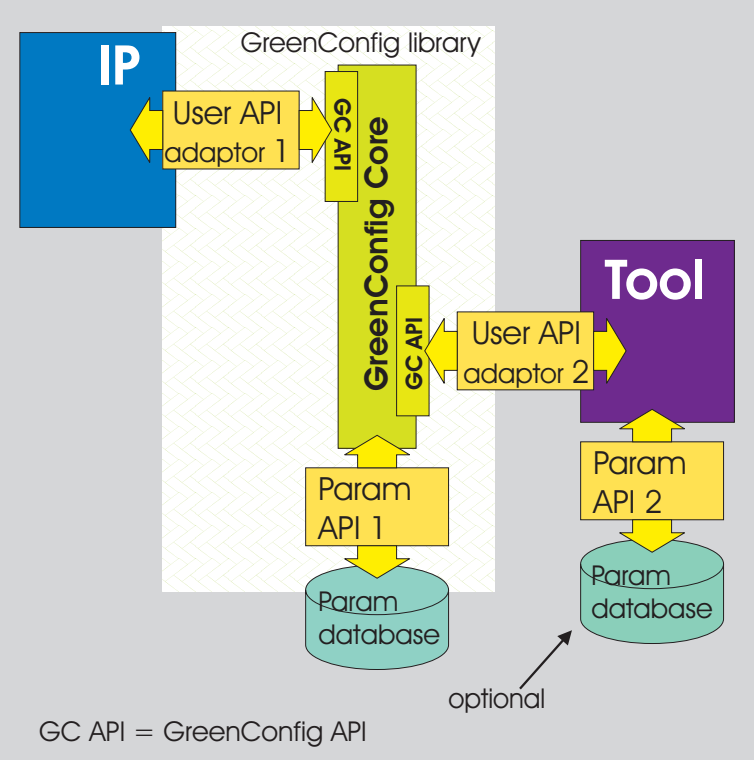

Figure A.2: Overview configuration with GreenConfig library.

A.1.3 Main requirements

- Support *most* config APIs (as much as reasonable regarding effort)
- Ease of use (i.e. usage requires minimal to no additional implementation effort)
- Extensibility (e.g., support more data types, add new config file formats, add additional functionality)

A.1.4 Requirements classification

We consider three classes of requirements:

1. Requirements for GREENCONFIG to support known configuration APIs
2. Functionalities we from our point of view would like to see in GREENCONFIG
3. Requirements for future directions

A.1.4.1 (1.) Requirements for GREENCONFIG to support other parameter configuration APIs

Table A.1 is based on the spreadsheet searching_configurations_in_SystemC.xls

These requirements (A.1.4.1) are made from the view of supporting the tool and user APIs of all other vendors.

- Provide an interface for the user and the tool (eventually uniform interface). This 'GREENCONFIG API' named interface is GREENCONFIG internal and is used by the specialized 'User API adaptor' for a special vendor. The GC API has to support all thinkable functionality of any user and tool API of any vendor.
- Interface to user API (GREENCONFIG User API adaptor 1, see figure A.1) must allow:
  - Registering a parameter (The user can add a parameter to the configuration framework.)
  - Set default value (Set value of a parameter during elaboration)
  - Get value (Get the initial value of a parameter; set by default value or by tool)
  - Change value during runtime (The user may change value of a parameter even during simulation runtime.)
  - Notify changes to user (Register observer for a parameter either a parameter of the own module or a parameter of another module.)

  Interface to tool API (GREENCONFIG User API adaptor 2, see figure A.1) must allow:
  - Configuration during elaboration / runtime (Set value of a parameter during elaboration and runtime)
  - Notify changes to tool (Register observer for a parameter which can be changed by the user during runtime)
  - Get value during elaboration / runtime (Get the value of a parameter during elaboration and runtime)
  - This interface may be used by a GUI tool
- The User API adaptors must be specialized for each vendor, using the GREENCONFIG API and providing the vendor's APIs.

	supported by				
	CoWare	**ARM**	**TI** no parameters	**Intel**	**GreenS. Simple Config. Framew.**
Data types	int, unsigned int, double, bool, std::string	std::string	unknown, however presentation implies that strings and number formats are supported	”Instrumented Datatype” user implemented implement interfaces	PODs, STL, SystemC
User defined types	-	yes (user implemented)		yes (instrumented data type)	yes (template specialized)
Permanent storage	XML	-	-	file	file
Registering a parameter	yes (global registry)	yes	-	drf module finds parameters itself	yes (`addParam` or macro `GB_PARAM`
Set default value	yes (constructor)	user may implement	user may implement	yes	yes (constructor)
Get value	yes (e.g. `getIntProperty( key)`	yes (`getParameter (string key)`)	yes (pushed by method call)	yes (Interface I_dr_dump)	yes (`get`)
Config. during runtime Done by other module or tool	-	-	yes (task_start)	-	yes (`set`)
Change value during runtime Done by module itself	yes	yes (user implemented)	yes (user implemented)	yes (user implemented)	yes (`set` or direct)

Table A.1: (a) This spreadsheet shows the capabilities of the reviewed configuration systems.

	supported by				
	CoWare	**ARM**	**TI** no parameters	**Intel**	**GreenS. Simple Config. Framew.**
Notify changes to user	-	-	yes (task call)	yes (event as member of instr. datatype)	-
Configuration during elaboration	yes (instantiation, with global registry)	yes (in user module `setParameter(` ↵ `string key, string` ↵ `value)`)	no (but task_start)	yes (Interface I_dr_config)	yes (constructor or `set`)
/ runtime	-	-	method calls	-	yes (`set`)
Notify changes to tool	-	-	task-finished event	-	-
Get value during elaboration	-	-	-	yes (dump)	-
/ runtime	-	-	-	yes (dump)	-

Table A.2: (b) This spreadsheet shows the capabilities of the reviewed configuration systems.

A.1.4.2 (2.) Requirements for GreenConfig in General from bottom-up

These requirements describe the framework from our view without regarding special other APIs but to achieve a many-sided framework.

1. *Usage*

   (1.1) New datatypes can be easily added by the user (to provide as many datatypes as possible)

   1.1.1. Methods to set and get the value of the parameter use `std::string`:
   void set(const std::string &str) and const std::string get()
   This allows easy adaption of the tools to set a parameter and is universal.

   1.1.2. The data type of a template should be given as template parameter.

   1.1.3. Usage of a templated parameter (`gs_param<datatype>`) should be as easy as the usage of the data type itself by overloading the operators (&, =). This is the instrumentation of a member variable to use it as a parameter. The usage of the parameter is transparent due to the overloaded operators. (SOW 3.1.3)

   (1.2) Support all imaginable parameter types and usages, e.g.

   - Setup parameters: Setup parameter is set during elaboration. The module complies with the value of the parameter to act variedly.
   - Change setup parameters: Parameter which changes during runtime to take influence on the behavior of the module.
   - Status parameters: Status parameter represent the status of a module. They are set by the module itself. Observers are notified when it changes.
   - Output parameters: Parameters to send output (debug values etc.) to the tool outside the simulation.
   - ...

2. *Configuration*

   (2.1) Configuration during elaboration

   (2.2) Configuration during runtime

   (2.3) Use same configuration system (same methods etc.) for initial configuration during elaboration and runtime configuration during simulation

   (2.4) Provide setting of parameters to other IPs (e.g. tools)

   - Setting of a parameter: `setParameter(std::string hierarchical_name, std::string value)`

   (2.5) Provide getting of parameter values to other IPs (e.g. tools)

   - Getting of a parameter value: `std::string getParameter(std::string hierarchical_name)`

   (2.6) Provide setting of parameters inside user code. There are two different types:

   2.6.1. Setting of the parameter in the module which contains the parameter:
   Inside the parameter holding module the user may choose out of two ways of setting the value.

- The user can set the value with the setting method (see requirement 1.1.1) using the parameter's string representation.
- The value can be set with the instrumentation of the member variable by simply assigning the value (see requirement 1.1.3).

2.6.2. Setting of the parameter of module in another module We have two options to realize the access to parameters of another module: First we can access them directly by pointer. This assumes the availability of the pointer which has to be provided by a central unit. It is clearer to provide the setting of a parameter by a function contained in the API of the library. Here the same method can be used which is also used by the tool API to set parameters (see requirement 2.4).

(2.7) A default value can be set during instantiation/construction of the parameter: `gs_param(const char *name, T value)`.

3. *Core*

(3.1) Efficient data management (database) for runtime (how parameters are saved during runtime, e.g. map, list,...)

(3.2) Efficient data management (database) for permanent storage (e.g. config file, XML file, Access database,...)

(3.3) Find local and global identification system for parameters (names): Unique name inside module, full name composed of hierarchical names leading to parameter name (`module_name.submodule_name.parameter_name`)

- Name a parameter during instantiation with `my_param = gs_param<datatype>(my_name)`.
- Get the parameter name with `const std::string &getName()`.

4. *Observation / notification*

(4.1) Provide a complete parameter list: e.g. `multimap<const char * hierarchical_prefix, const char * name> getParameterList()` or `list<std::string hierarchical_name> getParameterList()`

(4.2) Allow an observer to register for a parameter to be notified with an event when it changes

4.2.1. Allow many observers per parameter

4.2.2. Additional one event per configurable module `config_changed`?

(4.3) Provide to the module itself an event which is notified when any parameter of the module is changed (function callback *and* sc_event) (see requirement 4.2.2).

(4.4) Either the parameters have to register themselves at the library or the library finds the parameters or modules due to the implementation of the config interface (e.g. `gs_configurable`).

5. *Additional requirements resulting on conference call on 21st May 2007*

- Secure IPs which allow disabling their configuration abilities.
- Some kind of privacy tags to protect parameters being changes from outside the module.
- Recording actions (who is when setting or getting etc.) in the Core

A.1.4.3 (3.) Requirements for the support of other control interfaces

These requirements are formed by additional control issues. Here not the parameter configuration must be provided but more general control methods. E.g. configuration or starting a task with the call of a configuration method which is provided by a given or user defined interface .

This part is for future use and has no priority at this moment.

A.1.5 Sorted and prioritized requirements

1. Default value during construction
2. Configuration during elaboration
3. Configuration during runtime (Callback/event on change)
4. Supported data types (set and get with strings, addition of parameter types easy)
5. User defined data types
6. Notify changes to tool (Callback/event on change)
7. Notify changes to user module (Callback/event on change)
8. Permanent storage in file