GreenControl User's Guide (GreenControl v.4.3.0)

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

10th May 2011

Contents

List of Figures

Chapter 1

GreenControl Introduction

This section contains a brief overview of the implementation ideas for the *GreenControl* project. The GREENCONTROL framework is extendable with *plugins* which provide a service and add new functionality. To minimize the effect on analysis tools as little as possible SystemC elements are used within GREENCONTROL and its extensions (since v. 4.1.0).

See the project page[1] for further documentation and downloads.

1.1 Transaction-based approach

The GREENCONTROL framework is based on a transaction-based approach. The GREENCONTROL Core is a router that connects user modules with service providers (plugins).

- The connections are established via ports (but not using SystemC ports).
- Communication takes place using a special extendable transaction container.

Figure 1.1 shows the approach. The GREENCONTROL Core routes transactions between the IPs and plugins. One or more of the IPs may be a tool.

1.2 Transaction container

A transaction container contains generic attributes to transfer commands between the user modules and the service providers. The concepts of *atoms* and *quarks* in the GREENBUS transaction container will be adopted.

Example:

Service = ConfigService
Target = top.jpeg.compression_rate
Command = setParam
Value = 42
...

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

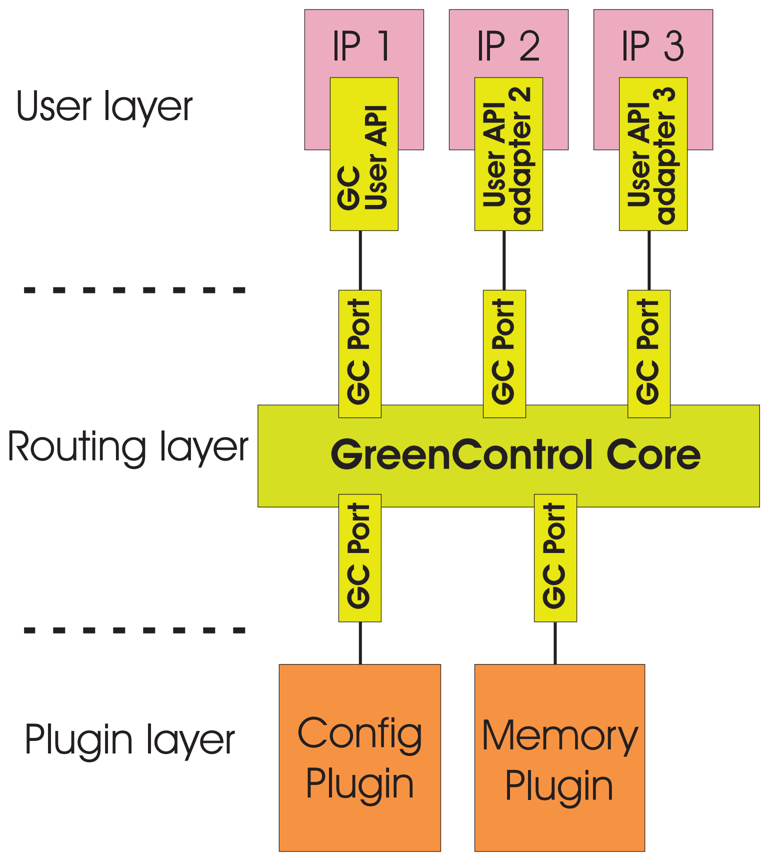

Figure 1.1: GreenControl framework with service plugins.

Advantages of this approach are

- decoupling of config APIs and config implementation
- well-defined protocol with clear semantics
- easy to add new commands
- easy to add new services, e.g. address management, memory framework, debug fabric, ...
- dynamic extension mechanism

1.3 GreenControl Core

The Core forwards transactions from the initiator to the target. For the above example, the target would be a ConfigPlugin. Upon receipt of the transaction, the ConfigPlugin would process the service request *setParam("top.jpeg.compression_rate", 42)* and acknowledge the transaction.

- Flexibility: Service providers (plugins) and user modules can be attached at elaboration time, without the need for re-compilation of the model.
- Reliability: if a service is requested that is not available/implemented by the service provider, the transaction can be rejected and a warning generated.

- Debugging: all transactions can be monitored in order to trace usage of the GreenControl services.

1.4 User APIs

User APIs are the APIs the user module sees and interacts with. They provide some functionality to the user module (IP). User APIs are connected to the GreenControl Core with a port (GC Port). They send transactions to the one service plugin which belongs to their functionality and task.

- APIs provide methods whose calls are translated into the appropriate GreenControl transactions.
  - Simple API methods such as setParam/getParam can be directly translated into a transaction.
  - More complex API method calls may require some housekeeping.
- User APIs receive transactions from the plugin and process them.

1.5 Plugins

Plugin are the service providers for the GreenControl framework. Different plugins may provide different functionality, such as configuration, analysis, visibility, memory and address management, debugging etc.

Thus, GreenControl is a versatile base fabric for the implementation of SystemC extension frameworks.

Existing plugins are the *configuration service plugin* GreenConfig[2] and the *analysis and visibility service plugin* GreenAV[3].

1.6 Additional information

The SystemC model only needs to include the used plugins. When e.g. memory management isn't needed, this plugin can be left out. The connection is done automatically during instantiation, hence no recompiling is needed if the usage of a plugin is changed.

The communication through ports and universal transaction containers gives the ability to extend GreenControl without changing the Core itself.

Newly developed plugins can be connected with the standard port without changing the GreenControl Core. So this extension can even done by a user.

To give the ability of extending GreenControl the transaction container has to be either very general to be useful in a new developed plugin or it has to be extendable itself. The extension of the container could be done by simple inheritance. The transmission of a transaction is more complex (and more time consuming) than simple method calls.

[2] GreenConfig project page: http://www.greensocs.com/projects/GreenControl/GreenConfig
[3] GreenAV project page: http://www.greensocs.com/projects/GreenControl/GreenAV

Chapter 2

GreenControl

2.1 Overview

See figure 2.1 to get an overview of the GreenControl classes and how they are connected.

2.2 Namespace gs::ctr and naming conventions

This framework is placed inside the namespace gs::ctr which is a sub namespace of the GreenSocs namespace gs. All GreenControl classes are placed in this namespace.

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

The classes that are visible to the user have the prefix `gc` or `GC`.

The GreenControl framework, and this documentation make use of some abbreviations:

- *GC* means *GreenControl*,

2.3 Files

The GreenSocs package *greencontrol* contains the framework files.

The framework is organized in subdirectories:

- greencontrol :
  Main user include files for the Core and services.
- greencontrol/core :
  GreenControl Core with all classes needed by the user to create the core and needed by the plugins and APIs for connection.
- greencontrol/<servicename> :
  For each service which can be added to the framework, a subdirectory is created which contains

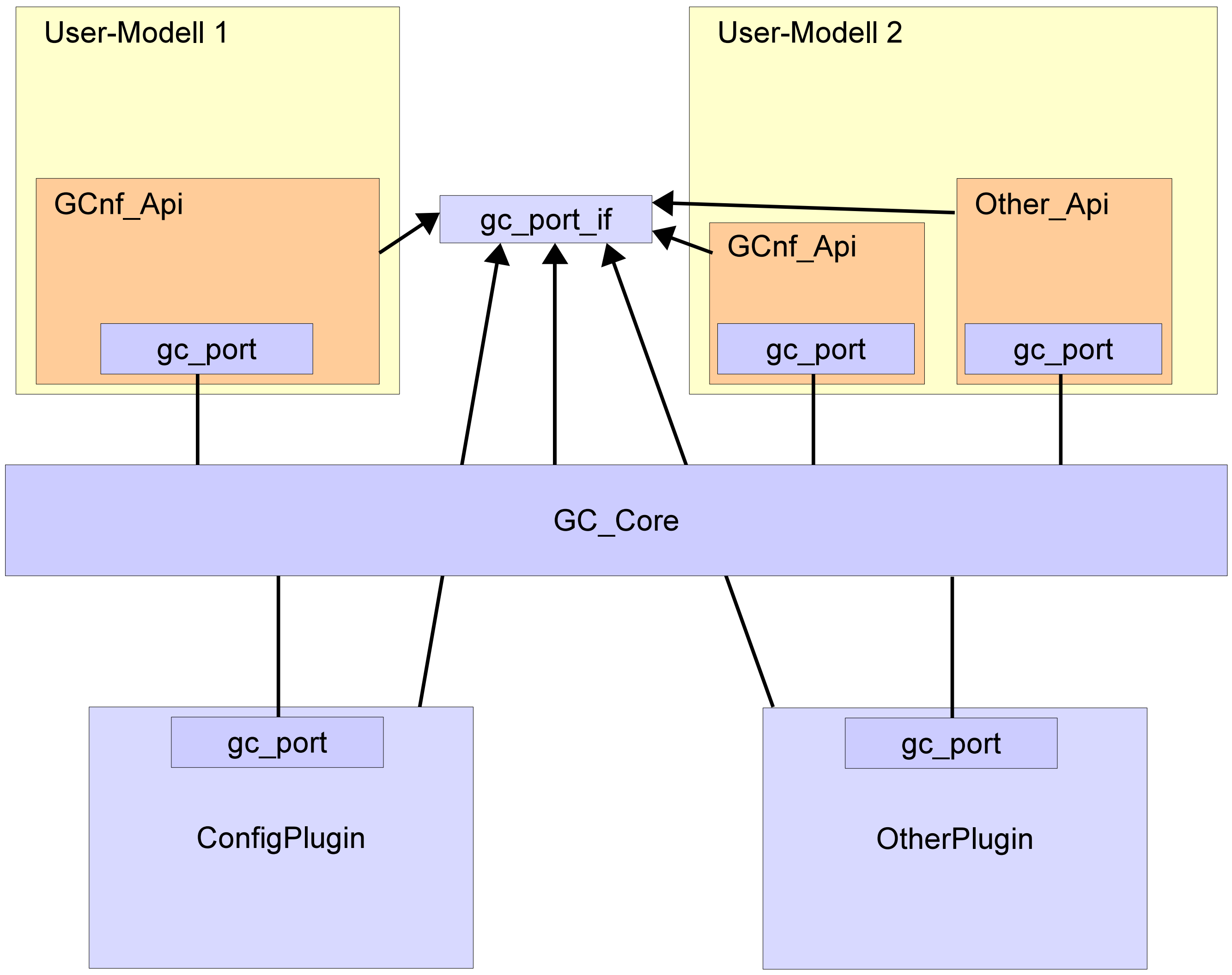

Figure 2.1: This image shows how the GreenControl implementation is realized.

  - plugin : a plugin directory and
  - apis : an API's directory

  Each service should provide a single include header file in the greencontrol folder. The file should include the basic classes needed for the service.

- greencontrol/docs/<servicename> :
  Documentation directory with one subdirectory for each service.

- greencontrol/examples :
  Examples

2.4 Communication

The GREENCONTROL communication mechanism uses a methodology similar to TLM, formally it was based on GREENBUS. It is important to notice that although the TLM concept of single transport calls transfering an extendible transaction container is used, no SystemC or OSCI TLM code is used, to avoid the GREENCONTROL mechansim from being visible in SystemC analysis tools.

2.4.1 Port binding

- An API or Plugin automatically connects to the Core just by instantiating a `gc_port`, which automatically performs the connection.
- The constructors of the control ports ask for special information:
  - APIs and plugins have to specify their supported control service.
  - Plugins have to set a bool to true to show that a plugin is connected to serve all transactions to its supported service.
  - APIs and plugins have to specify their target address (type `cport_address_type`) which is a `gc_port_if` pointer and has to be set to their pointer address.
  - APIs and plugins submit a human readable unique name which is used for debug and maybe future use.
  - These variables are read by the Core when building the address map (see section 2.4.2).

2.4.2 Addressing

The GREENCONTROL Core routes the transaction containers with the two fields ”Service” and ”Target”. The target field is the address (target's `gc_port_if` pointer address) of the API or plugin where the transaction should be delivered. The service field specifies the service which is used for categorization (for analyzing the communication) and is used for routing:

If the sender does not know the correct address of the plugin which is responsible for the service (normal case!), the sender may leave the target field blank (`0`) and only set the service field. The router knows which plugin belongs to which service (because of the constructor parameters of the ports, see section 2.4.1) and routes the transaction to the (only) plugin supporting that service.

The plugin has to know the address of the API it wants to reach. That is no problem because each API first sends a transaction to the plugin (e.g. parameter add, set or observer registration) and the plugin stores the MasterID (which can be found in the transaction container) as target address for that API/module).

The target address allows a user module to use multiple (different) APIs for one service (e.g. it uses scml parameters and GCnf parameters).

2.4.3 Port usage in the APIs

This is about how to use the ports inside an implementation of an User API adapter.

The interface `gcnf_api_if` waits for the method:

- `void transport(ControlTransactionHandle& tr)`

This method is called by the sending initiator.

2.4.4 Transaction Container

The transaction container consists of various fields (see table 2.1 on page 11).

The command field is kept general as an unsigned integer. Since the transaction object `ControlTransaction` could be used by various services this field is dependent on the specific service being used. The APIs and plugins should use an enumeration which is specialized for their service.

Not all fields are necessarily used by each service. Since we have a pool of existing transaction objects and not all fields are reseted each time being reused there is no performance issue having more fields than needed.

The sender of a transaction *must ensure* that *all* fields that are processed by the receiver are set or reseted because transactions are taken out of the pool and may contain not reseted fields! The only fields that are reseted by default are the Error field and the Service field.

See GREENCONFIG User's Guide for the available commands for the configuration service and how to add new commands for that service.

2.4.5 Transaction Extensions

The transaction class provides an extension mechanism. This mechanism is similar to the TLM-2.0 extensions.

There is a base class `gc_transaction_extension` to derive user extensions from to store and transport them in the transaction. Derive your transaction extension class in the following way:

Field name	*Member name*	*Data type*	*Example 1*	*Example 2*	*Explanation*
Service	mService	ControlService (enum)	`CONFIG_SERVICE`		Service specification; used for routing to the correct plugin and analysis
Target, Address	mTarget	cport_address_type (void*)	`3215719780`	`3215759780`	Routing target (address of API or plugin) where the Core should route the transaction to
Command	mCmd	unsigned int (for specialized enumerations)	`CMD_SET_PARAM`	`CMD_GET_PARAM_LIST`	Command the target should execute, available commands are dependent on the service.
AnyUint	mAnyUint	unsigned int			Field for any type of unsigned int information.
Specifier	mSpecifier	ControlSpecifier (string)	`jpeg.` ↩ `compressionParam1`	`mymodule.submodule.` ↩ `param2`	The affected object; different concerning to the command
Value	mValue	ControlValue (string)	`"42"`	`"myTestValue"`	Value; different concerning to the command
MasterID	mID	cport_address_type (void*)	`3213638188`		Address of the sender API or plugin. Automatically set by port.
Error	mError	unsigned int	`0`	`1`	Error code for response (0=no error)
LogPointer	mLogPointer	log_if*			Field for any type of pointer, with logger information.
AnyPointer	mAnyPointer	void*			Field for any type of pointer, initiator and target must know the type and do casts. Better use LogPointer if possible!
AnyPointer2	mAnyPointer2	void*			Another field for any type of pointer. Better use LogPointer if possible!

Table 2.1: Fields of the Transaction Container.

```
class my_extension
  : public gs::ctr::gc_transaction_extension<my_extension>
  , public gs::ctr::log_if
```

The base class `log_if` is optional but recommended to allow logging with the GC_Logger!

Extensions can be added to a ControlTransaction (one of each extension type), similar to TLM-2.0-extensions but not using memory management. Extension can only cleared (not released) because the owner shall care about the extension's memory management - which is not hard because a `ControlTransaction` shall not be assumed to live longer than just during the transport call (this is different from TLM-2.0).

No `ControlTransaction` should be created before static initialization ended, because the maximum number of extensions may not be known yet before! If a transaction is created during static initialization, its extensions vector must be resized to `max_num_gc_transaction_extensions()`.

Typically it is not necessary to copy transactions. When copying transactions, no deep copy of pointers and extension is made!

⚠ Note: There are two different extension types: Transaction Extensions (these ones) and Service Extensions (see section 2.8)!

2.5 Order for constructing GreenControl elements

The GREENCONTROL framework requires the a strict constructing order. It is recommended to instantiate the elements manually observe the following rules. Alternatively the automatic mechanism described afterwards can be used if there is no way to ensure that the GREENCONTROL elements are instantiated before any code accessing the services, e.g. in a tool environment.

- Before any other GREENCONTROL elements may be instantiated or used, the singleton *Core* is needed.

  ```
  gs::ctr::GC_Core   core;
  ```

  Alternatively use the following static function call which will create (at first call) and return the singleton (each call):

  ```
  gs::ctr::GC_Core::get_instance();
  ```

- Afterwards other GREENCONTROL elements may be instantiated, beginning with the plugins. For example the Configuration Service within the namespace `gs::cnf` is the next one to instantiate:

  ```
  gs::cnf::ConfigPlugin   configPlugin(database);
  ```

  See the related documentation for the recommended and possible ways of creating the service plugins.

- The plugin instantiations may be followed by any User APIs and modules (containing User APIs).

The construction order for plugins, APIs and modules is not fixed. Depending on the application different orders may be useful. In general it will be reasonable to instantiate all plugins prior to modules which make use of them. If a user module tries to make use of a not existing service plugin during its construction the routing of the transaction will fail and result in a routing error.

2.5.1 Alternative automatic construction

Alternatively to the manual Core instantiation, you can rely on its automatic creation. During the first creation of a `gc_port` the Core will be created by the static get instance function being called by the port constructor. Typically this should be the case when creating the first service plugin, e.g. the configuration plugin.

See the related documentation for possible automatic instantiations of the plugins.

2.6 Command line options/switches

Some API Tools may need to parse command line options and switches, e.g. config file parser to get the filename (see GREENCONFIG documentation), config command line parser to get parameter values etc.

To get a variable framework where these APIs and tools can be added and removed very easily we have to use a general approach for parsing the command line:

To achieve highly modularity each API tool, which needs to parse the command line, gets the whole command line arguments like they are submitted to the `main` method. The parser then only processes the options and switches which are supported by that parser and ignores the other ones. The parsers must not change the `argc` value and the `argv` array because all parser should get the original values. The parser should work on a copy of the argument array: if e.g. the command `getopt` or `getopt_long` is used, a copy of `argv` has to be used because these methods change the arrays during processing.

- It is essential to *make sure that each option or switch is used only once in the entire framework.*
- The parsers may make use of (and throw or throw and catch) the exception class `CommandLineException`.
- All parsers should parse the '–help' and '-h' switch to print out usage information.
- None of these parsers is allowed to make use of non-option and non-switch arguments because they are printed out by the core to allow a minimal user information on wrong used arguments: The switches and options are parsed by the concerned parser and remaining elements (due to wrong placed spaces etc.) should be printed out as failures.

2.7 Initialize-mode

The GREENCONTROL framework provides the *initialize-mode* which can be used by plugins and APIs to differ between elaboration and runtime without the need to derive from SystemC modules.

GREENCONTROL may be used immediately after construction (even before simulation runtime).

To give the APIs the opportunity to check if the time is during initialize-mode there are callbacks:

If the APIs had to override the kernel callback functions to make their elaboration configuration they would be forced to be SystemC modules. But we don't want them to be modules. So we have our own interface (`initialize_if`) which has to be implemented by each API which wants to use the initialize-mode.

All implementations of this interface are called by the only `sc_module` in GREENCONTROL which is the so-called `CallbackDispatcher`. The interface automatically registers itself with the Core (and the dispatcher), and unregisters as well when being destructed. So simply deriving from this interface allows to use the functions and being called.

The interface contains the callback functions `start_initialize_mode` and `end_initialize_mode`. Both functions are called by the Core while it processes the start_of_simulation SystemC callback. The first callback (`start_initialize_mode`) may be used for initial configuration which was not yet done at construction time. The latter (`end_initialize_mode`) shows the end of initialize-mode. From that point on the framework may be used in *runtime-mode*. With the help of these callbacks plugins and APIs can identify simulation runtime. Before and during the initialize-mode *no* events, notifies, time (even SC_ZERO_TIME) are allowed within the service communication. There are also optional calls `gc_before_end_of_elaboration`, `gc_end_of_elaboration` and `gc_end_of_simulation`.

An example how to use the initialize-mode callbacks can be seen in the GREENCONFIG API implementation.

2.8 New Services

For adding Services to GreenControl there is an extension mechanism.

The base class `gs::ctr::gc_service_ID_extension<my_service_ID>` is to derive service extensions from. Use the base class to identify the Plugin and API(s) against the `gc_port`. The extension mechanism will automatically care for a simulation-wide unique ID[1].

Derive your service extension class in the following way:

```
struct test_service_ID
: public gs::ctr::gc_service_ID_extension<test_service_ID>
{
  test_service_ID()
  : gs::ctr::gc_service_ID_extension<test_service_ID>("TEST_SERVICE")
  { }
};
```

The extension constructor defines the debug and log string connected to the service ID.

This triggers proper service extension registration during C++ static contruction time. There will be generated a unique Service ID.

[1]Note for debugging: This unique Service ID may vary when the number of used Services is changed or even if the include or instantiation order is changed!

⚠ Note: There are two different extension types: Service Extensions (these ones) and Transaction Extensions (see section 2.4.5)!

2.8.1 How to create new Service Plugins and User APIs

See the example new_service_example for an example how to develop new Service Plugins and User APIs. Additionally you should have a look at the GC_Logger which helps to analyse communication[2] traffic over the Core.

2.9 Miscellaneous

2.9.1 SystemC 2.1

When using GREENCONTROL with SystemC 2.1 there may be warnings like the one below. For OSCI SystemC 2.1v1 there is a workaround. Within other SystemC implementations these warnings can be presumably ignored.

```
Warning: (W505) object already exists: fec. Latter declaration will be renamed to fec_24
In file: ../../../../src/sysc/kernel/sc_object.cpp:187
```

[2]The GC_Logger currently does not support logging of transaction extensions