

Preventing Wheel Reinvention: The psgconf System Configuration Framework

Mark D. Roth – University of Illinois at Urbana-Champaign

ABSTRACT

Most existing Unix system configuration tools are designed monolithically. Each tool stores configuration data in its own way, has its own mechanism for enforcing policy, has a fixed repertoire of actions that can be performed to modify the system, and provides a specific strategy for configuration management. As a result, most tools are useful only in environments that very closely match the environment for which the tool was designed. This inflexibility results in a great deal of duplication of effort in the system administration community.

In this paper, I present a new architecture for system configuration tools using a modular design. I explain how this architecture allows a single tool to use different strategies for configuration management, enforce different ideas of policy, and prevent duplication of effort. I also describe the implementation of this architecture at my site and identify some areas for future research.

Introduction

Most Unix system configuration tools are composed of several common types of components:

- **Data Store:** The data store is the repository from which the tool reads the configuration data explicitly supplied by the system administrator. For example, ISConf [1] uses Makefiles, LCFG [2] uses X resources-style source files, cfengine [3] uses its own custom configuration file format, TemplateTree II [4] uses template files that are used to generate a cfengine configuration file, Arusha [5] uses XML files, and Simon [6] stores configuration data in an SQL database.
- **Policy Rules:** Policy is the programmatic manipulation of configuration data. Unlike the explicit configuration parameters read from the data store, policy rules act implicitly to enforce requirements imposed by the administrator. For example, at some particular site, “enable anonymous FTP” might automatically mean to install `wu-ftpd`, create the `ftp` user, add the appropriate entry to `inetd.conf`, update TCP wrappers, etc.
- **Actions:** Actions are the mechanisms for manipulating the system’s state. For example, cfengine supplies specific file-editing actions like `AppendIfNoSuchLine`, while almost all existing tools allow you to execute arbitrary shell commands.

Despite these commonalities, most tools are very different from one another in the way that they put their components together to form a general strategy for configuration management. For example, ISConf takes the extreme approach of stepping through each state in a machine’s configuration history in order to get to the current/desired state; cfengine is designed to converge from what a given file looks like now to what you want it to look like; and Simon is designed to generate individual configuration files from scratch.

Unfortunately, most existing tools are designed monolithically, which means that you cannot choose the components or strategy independently. For example, there’s no way to use cfengine’s convergence strategy with Simon’s SQL data store. There’s also no way to use Simon’s file generation strategy for some configuration files and cfengine’s convergence strategy for others.

This inflexibility results in a great deal of duplication of effort in the system administration community, because the only way to change out a single component or add support for a different strategy is to write a whole new tool. The new tool may include new components or support a different strategy, but a great deal of time is also spent duplicating existing components from other tools.

A New Approach: The psgconf System

Here at the University of Illinois at Urbana-Champaign (UIUC), I’ve developed a tool called psgconf to address these problems. Instead of being a monolithic tool with a fixed set of components, psgconf dynamically loads external modules that implement its components. This makes it very easy to add or remove components as needed.

The psgconf system is written entirely in Perl, and its components are nothing more than Perl objects that provide the appropriate methods for the component type. This means that writing a new component for psgconf is as simple as writing a Perl module that provides the component’s object class.

Object Structure

Each of the next subsections describes an object that is part of the psgconf system.

The PSGConf Object

The central object in the psgconf system is provided by the PSGConf module. The PSGConf object is extremely simple; its primary function is to coordinate the activity of the other objects, which are where most of the work is actually done.

Data Objects

Instead of storing configuration data in static variables, psgconf encapsulates data in Data objects. This provides a great deal of flexibility in configuring the system, because each Data object class can provide whatever methods are appropriate for the encapsulated data.

All Data object classes should be derived from the PSGConf::Data base class. The base class provides several fundamental methods, such as set(), get(), and unset(). However, new subclasses are free to override these methods or to add whatever new methods are appropriate for the type of data the class intends to represent. The psgconf system includes several such modules that provide useful Data object classes:

Data Object Class	Description
PSGConf::Data::Boolean	boolean data
PSGConf::Data::Hash	hash table data
PSGConf::Data::Integer	integer data
PSGConf::Data::List	list data
PSGConf::Data::String	character string data
PSGConf::Data::Table	table-oriented data

For example, the PSGConf::Data::String module provides a Data object class for character-string data. It provides several methods that are specially tailored for manipulating string data: append(), which appends its argument to the end of the string already contained by the object; prepend(), which prepends its argument to the beginning of the string already contained by the object; and gsub(), which replaces substrings in the string already contained by the object.

As each Data object is instantiated, it is registered with the central PSGConf object under a particular name, which can then be used to reference the object. For example, there might be a PSGConf::Data::String object called log_dir or a PSGConf::Data::Boolean object called anon_ftp_enable.

DataStore Objects

In the psgconf framework, a system's configuration is expressed as a series of configuration statements

that manipulate the registered Data objects. Each statement consists of the name of a Data object, a method to call on that object, and a list of arguments to pass to that method. The mechanism for reading these configuration statements is provided by DataStore objects.

Each DataStore object provides a method called read_config() that reads configuration statements from some arbitrary source and executes the requested method calls. It might read the statements from a local configuration file using some particular syntax, or it might use a network protocol to read the statements from a remote configuration server (referred to as a "gold server"). It does not matter to psgconf how the statements are actually read, as long as the DataStore module can read the statements and execute the requested method calls.

For example, the PSGConf::DataStore::ConfigFile DataStore object reads configuration statements from a local file using syntax that looks like this:

```
log_dir->set "/var/log";
```

This statement tells the DataStore object to look for a Data object named log_dir and call that object's set() method with the string /var/log as its argument.

Action Objects

Action objects represent actions to be performed on the system.

All Action object classes should be derived from the PSGConf::Action base class, which provides several support methods. However, new subclasses should always define three important methods: check(), which determines whether the action actually needs to be performed or whether the system is already in compliance; diff(), which displays details about the changes that would be made to the system; and do(), which actually performs the action.

The psgconf system includes many useful Action classes. See Table 1 for several examples. Action objects are processed in the order in which they are registered. This is discussed in more detail below.

Control Objects

Control objects are the real workhorses of the psgconf system. They are responsible for directing the overall process of configuring the system.

From their constructor, Control objects can instantiate and register new Data objects. As mentioned above, each Data object is given a name as it is

Action Object Class	Description
PSGConf::Action::GenerateFile	programmatically generate a file
PSGConf::Action::MkDir	create a directory
PSGConf::Action::ModifyFile	programmatically modify an existing file
PSGConf::Action::RunCommand	run an external command
PSGConf::Action::Symlink	create a symbolic link
PSGConf::Action::TouchFile	create an empty file

Table 1: Examples of Action classes.

registered. For example, the `PSGConf::Control::AnonFTP` module registers the Data objects shown in Table 2.

Control objects can also provide any number of policy methods, which perform programmatic manipulation of Data objects. As with Data objects, the Control module's constructor registers each policy method under a particular name. Going back to the previous example, the `PSGConf::Control::AnonFTP` module might register the policy methods in Table 3. Note that just because a policy method is registered does not necessarily mean that it will be used. This is discussed in more detail below.

Control objects also provide a method called `decide()`, which is responsible for instantiating and registering new Action objects based on the final values of the Data objects. For example, the `PSGConf::Control::inetd` module's `decide()` method checks whether the `inetd` Data object contains any entries, and if so it instantiates a `PSGConf::Action::GenerateFile` object to create the `inetd.conf` file.

The Big Picture

To illustrate how all of these object types fit together, it is useful to enumerate the steps taken by `psgconf` to configure the system.

Step 1: Control and DataStore Module Instantiation

The `PSGConf` object starts by reading the `/etc/psgconf_modules` file, which can contain the following types of entries:

```

DataStore module_name [args ...]
Control   module_name [args ...]
Policy    method_name
    
```

The `DataStore` entries specify a `DataStore` module to be instantiated. Any additional arguments are passed to the module's constructor. Multiple `DataStore` entries can be present, in which case the modules are accessed in the order in which their entries appear in the `/etc/psgconf_modules` file.

The `Control` entries specify a `Control` module to be instantiated. As with `DataStore` entries, any additional arguments are passed to the module's constructor. Multiple `Control` entries can be (and usually are) present, in which case the modules are accessed in the order in which their entries appear in the `/etc/psgconf_modules` file. As mentioned above, the constructor for each `Control` object can instantiate and register any needed Data objects, as well as registering any policy methods it provides.

The `Policy` entries specify the set of policy methods that will actually be invoked. The methods will be

Data Object	Class	Description
<code>anon_ftp_enable</code>	<code>PSGConf::Data::Boolean</code>	whether or not to enable anonymous FTP
<code>anon_ftp_dir</code>	<code>PSGConf::Data::String</code>	path to ftp root
<code>anon_ftp_options</code>	<code>PSGConf::Data::Hash</code>	options for ftpaccess file

Table 2: Data objects registered by the `PSGConf::Control::AnonFTP` module.

Policy Method	Description
<code>anon_ftp_add_user</code>	if <code>anon_ftp_enable</code> is set, create ftp user
<code>anon_ftp_add_inetd_entry</code>	if <code>anon_ftp_enable</code> is set, add <code>inetd.conf</code> entry
<code>anon_ftp_add_package</code>	if <code>anon_ftp_enable</code> is set, install <code>wu-ftpd</code>

Table 3: Policy methods registered by the `PSGConf::Control::AnonFTP` module.

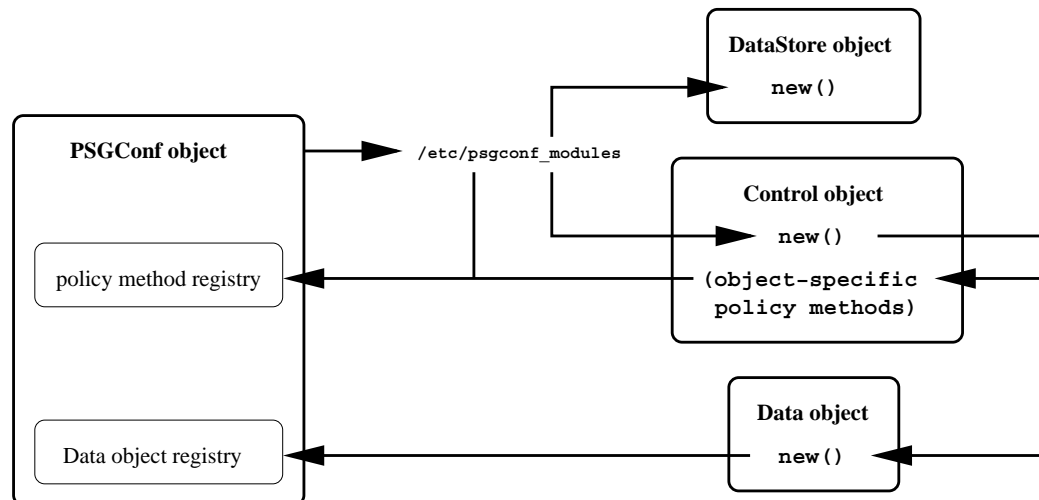


Figure 1: Control and DataStore module instantiation.

invoked in the order in which their entries appear in the `/etc/psgconf_modules` file. Note that as `psgconf` reads each Policy entry, it will verify that the specified policy method is actually registered. This means that it is an error to specify a Policy entry for a policy method before the Control entry for the Control module that provides that policy method. By convention, this problem should be avoided by placing all Policy entries at the end of the file.

Step 2: DataStore Processing

The PSGConf object loops through the list of DataStore objects specified in the `/etc/psgconf_modules` file and calls the `read_config()` method of each object. The `read_config()` method accesses the data store and reads configuration statements. Each configuration statement results in calling a method of one of the Data objects that was previously registered by the Control modules.

Step 3: Policy Enforcement

The PSGConf object calls each of the policy methods specified in the `/etc/psgconf_modules` file. The

policy methods programmatically manipulate Data objects to enforce policy.

Step 4: Action Instantiation

The PSGConf object loops through the list of Control objects specified in the `/etc/psgconf_modules` file and calls the `decide()` method of each object. The `decide()` method instantiates Action objects to perform the appropriate actions based on the content of the Data objects.

Note that the Action objects are processed in the order in which they are registered. Because they are registered by a Control object's `decide()` method, this means that the order of the Control objects in the `/etc/psgconf_modules` file actually determines the order of the Action objects instantiated by the various Control modules.

Step 5: Action Checking

The PSGConf object loops through the list of Action objects and calls the `check()` method of each object. The `check()` method checks to see if the action actually needs to be performed.

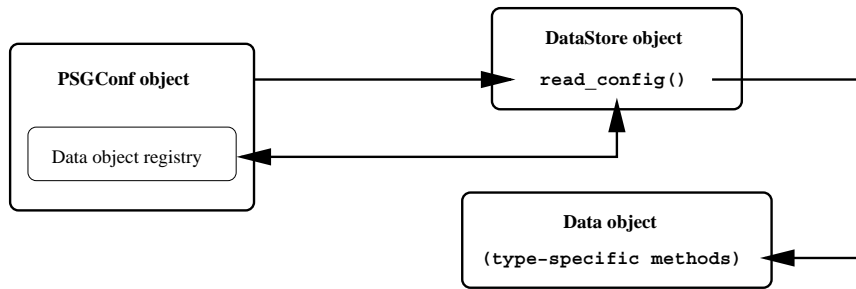


Figure 2: DataStore processing.

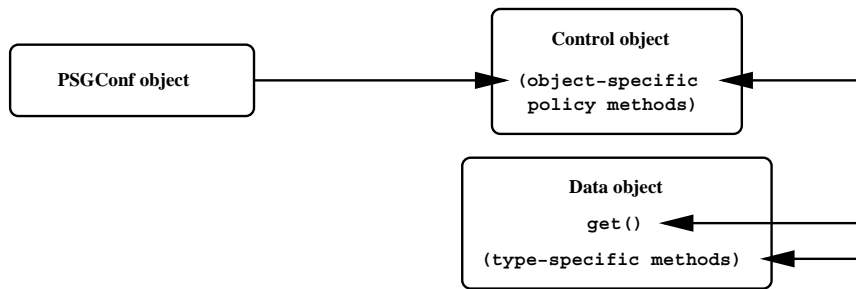


Figure 3: Policy enforcement.

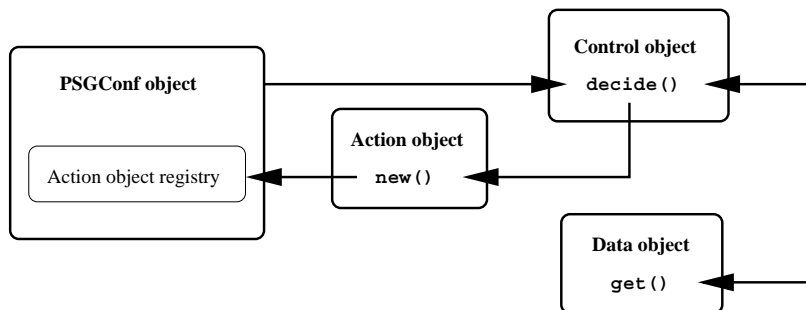


Figure 4: Action instantiation.

For example, the `PSGConf::Action::GenerateFile` module's `check()` method will generate the new file and compare it to the existing file; if the two files differ, then the action needs to be performed.

Step 6: Action Implementation

For each Action object that needs to be performed, the `PSGConf` object calls the object's `diff()` and/or `do()` method, depending on what options `psgconf` was invoked with.

The Action object's `diff()` method prints the details of the change that will be made to the system. For example, the `PSGConf::Action::GenerateFile` module's `diff()` method invokes the `diff(1)` command to show the differences between the existing file and the new file, which was generated by the `check()` method in the previous step.

The Action object's `do()` method actually performs the change on the system. For example, the `PSGConf::Action::GenerateFile` module's `do()` method replaces the existing file with the newly generated version.

Step 7: Cleanup

The `PSGConf` object loops through the list of Control objects and calls the `cleanup()` method of each object, if present. The `cleanup()` method performs any necessary cleanup tasks, such as restarting a daemon after its configuration file has changed.

The Dual Role of Control Modules

In practice, Control modules are written to serve one of two distinct roles: they can encapsulate the configuration of a particular subsystem, or they can provide features that span multiple subsystems.

Most Control modules are designed to encapsulate the configuration of a specific subsystem. For example, the `PSGConf::Control::AnonFTP` module

provides all of the necessary Action objects for creating the anonymous FTP tree, generating the `/etc/ftpaccess` file, and so on. Essentially, it contains everything necessary for configuring `wu-ftpd`.

Control modules can also be designed to provide subsystem-independent features. For example, my group uses a locally written Control module called `PSG::Control::ConnectionLog` that configures TCP wrappers to log connection information via `syslog` to a file called `connections`. It does this via a policy method that manipulates Data objects provided by the `PSGConf::Control::syslog` and `PSGConf::Control::TCPWrappers` modules. However, the `PSG::Control::ConnectionLog` module does not register any Action objects of its own, because it does not configure any subsystems directly.

Although these two roles are often distinct, they are both provided by the same type of module (Control modules) because there is a great deal of overlap in the mechanisms they use. Both may need to provide Data objects to make the relevant subsystem or feature configurable, and both may need to provide policy methods to manipulate Data objects provided by other modules. Rather than duplicate all of this functionality in two different module APIs, `psgconf` is able to handle both roles through the same interface.

Ordering Policy Methods

The order in which policy methods are invoked is important, because a given policy method may depend on changes made by another policy method.

For example, the `PSGConf::Control::AnonFTP` module provides a policy method called `anon_ftp_add_tcpd` that adds an entry to the `tcp_wrappers` Data object for the FTP server. In addition, my group's locally written `PSG::Control::ConnectionLog` module (described above)

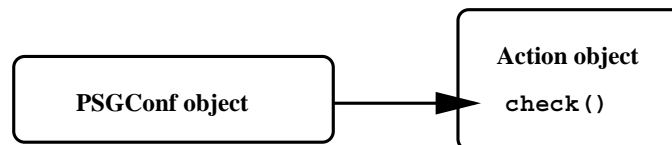


Figure 5: Action checking.

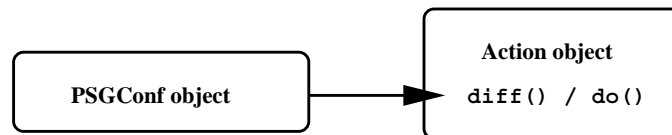


Figure 6: Action implementation.

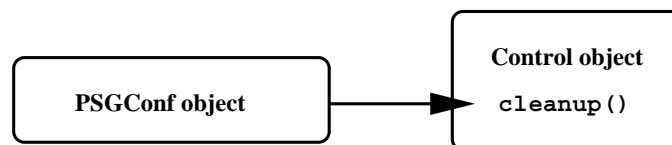


Figure 7: Cleanup.

provides a policy method called `connection_log_modify_tcpd`. This policy method checks the `syslog` Data object (provided by `PSGConf::Control::syslog`) for a logfile called `connections`. If found, it adds a `severity` option to each entry in the `tcp_wrappers` Data object so that wrapper information is sent to the `connections` log.

When the `connection_log_modify_tcpd` method is invoked, it adds the `severity` option for each entry that it finds. However, if the entry for the FTP server is not added until after the `connection_log_modify_tcpd` method runs, then the `severity` option will not be added to that entry. Therefore, the `anon_ftp_add_tcpd` policy method must be invoked before the `connection_log_modify_tcpd` policy method.

At first, I considered implementing a mechanism that would automatically determine the order in which to invoke the policy methods based on dependency information encoded into the Control module by the module's author. However, the module's author doesn't have any way of knowing what other modules or policy methods the module will be used with, so there's no way to encode all of the dependencies into the module. Only the administrator can do that, because he's the one choosing the Control modules and policy methods.

As a result, `psgconf` allows the administrator to set the order in which policy methods are invoked in the `/etc/psgconf_modules` file.

A Note About Object-Orientation

Note that I chose to use object-oriented modules for three main reasons. First, that's the established convention for writing new Perl modules. Second, the object-oriented approach provides an easy way for each object to maintain its own state. This is especially important for Data and Action objects, because multiple independent objects are often instantiated from each class. Finally, the object-oriented approach makes it possible to create new modules as subclasses of existing modules, which can make it easier to write and maintain variations of existing modules.

Advantages of psgconf

The `psgconf` system is extremely flexible. Several of its more noteworthy advantages are described here.

Flexible DataStore Mechanism

Because `psgconf` is not tied to a single DataStore implementation, it can grow with its environment as the environment's needs change. For example, the only existing DataStore implementation is the `PSGConf::DataStore::ConfigFile` module described above, which reads configuration data from a local configuration file. In the future, I plan to develop new data store modules to access configuration data from a central gold server using mechanisms like SQL queries or XML-RPC calls. Using these new data store modules will be as simple as updating the `/etc/psgconf_modules` file.

A Smorgasbord of Strategy

Because new Action classes can be created as needed, `psgconf` is not limited to a specific strategy for configuration management. For example, `cfengine`'s convergence strategy can be implemented for certain files using objects of the `PSGConf::File::ModifyFile` class, while other files can be generated from scratch using `PSGConf::File::GenerateFile` objects. Because Action objects are processed in a specific order, even `ISConf`'s history-recreation strategy can be implemented, simply by adding a new Control module for each state in the configuration history.

Separation of Config Data from File Formats

Because configuration data is represented in Data objects and specific file formats are understood by particular Action and Control objects, there is a clean separation of code and data. This makes it very easy to support platforms that each use a different file format to represent the same data. For example, the `PSGConf::Control::PAM` module provides a single Data object to contain all of the PAM configuration data, but it can instantiate Action objects to generate either `/etc/pam.conf` or individual files in `/etc/pam.d`, as appropriate. Similarly, a module might instantiate Action objects to generate either `inetd.conf` or `xinetd` configuration files based on the same Data objects.

Future Directions

There is much potential for improvement in the area of communication between package management tools and configuration management tools. Currently, using a `psgconf` Control module for something like Apache may require the administrator to override a lot of defaults, because the module does not know what default paths or features were built into the Apache package at compile time. If this information were encoded into the Apache package in some standard way, the `psgconf` module could query the package manager to get this information.

There is currently no mechanism for asynchronously notifying `psgconf` of changes in a central data store. As new types of DataStore modules are developed, this functionality will be implemented.

Although `psgconf` is publicly available, it has never been announced in any major public forums, and I am not aware of any other large sites using it. I would very much like to get feedback from others who are using it, so that any lingering features or assumptions that might be specific to my site can be eliminated.

Ultimately, I would like to create a CPAN-style site for distributing `psgconf` modules written by different people. The goal of this site would be to allow system administrators in different organizations to share `psgconf` modules, thus avoiding the need to reimplement a module that's already been written.

Conclusion

The psgconf framework's modular architecture provides a great deal of flexibility. Components can be swapped out to meet changing needs, new platforms can be supported without changing the configuration data model, and different strategies can be combined to manage system configuration in the most flexible manner. Eventually, I hope that the ability to share modules will help prevent wheel reinvention in the entire system administration community.

Availability

The psgconf package can be downloaded from its home site at <http://www-dev.cites.uiuc.edu/psgconf/>. It is distributed under a BSD-style license.

Acknowledgments

I would like to thank Paul Anderson, Alva Couch, Daniel Hagerty, Luke Kanies, Adam Moskowitz, and many others for providing frequent sanity checks as I was developing the psgconf framework.

I would also like to thank Steve Traugott for helping the system administration community think about infrastructure in a whole new way.

Author Information

Mark Roth is the technical lead of the Production Systems Group of the Campus Information Technologies and Educational Services department at the University of Illinois at Urbana-Champaign. Mark is the author of several open-source software packages. He can be contacted via email at roth@uiuc.edu, and his web page is <http://www.uiuc.edu/ph/www/roth>.

References

- [1] Traugott, Steve and Joel Huddleston, "Bootstrapping an Infrastructure," *LISA XII Proceedings*, Boston, MA, pp. 181-196, 1998.
- [2] Anderson, Paul, "Towards a High-Level Machine Configuration System," *LISA VIII*, Berkeley, CA, pp. 19-26, 1994.
- [3] Burgess, Mark, "Cfengine: A Site Configuration Engine," *USENIX Computing Systems*, Vol. 8, Num. 3, <http://www.cfengine.org/>, 1995.
- [4] Oetiker, Tobias, "TemplateTree II: The Post-Installation Setup Tool," *LISA XV Proceedings*, San Diego, CA, pp. 179-186, 2001.
- [5] Holgate, Matt and Will Partain, "The Arusha Project: A Framework for Collaborative Unix System Administration," *LISA XV Proceedings*, San Diego, CA, pp. 187-198, 2001.
- [6] Finke, Jon, "An Improved Approach for Generating Configuration Files from a Database," *LISA XIV*, New Orleans, LA, pp. 29-38, 2000.

