Configuration For GNU Project Summary and Documentation Guide

Jason Long

Justin Yackoski

Follow this and additional works at: https://mosaic.messiah.edu/honors

Part of the Computer Sciences Commons

Permanent URL: https://mosaic.messiah.edu/honors/269

Sharpening Intellect | Deepening Christian Faith | Inspiring Action

Messiah College is a Christian college of the liberal and applied arts and sciences. Our mission is to educate men and women toward maturity of intellect, character and Christian faith in preparation for lives of service, leadership and reconciliation in church and society.
Project Summary

Overall, our project was very successful and we accomplished a significant amount of both design and implementation work. The basic concept and architecture we designed during the fall semester has so far proved viable and advantageous over existing solutions. We created several proof-of-concepts applications and much of the core feature set is in place. We both hope to continue building on the project after we graduate.

We met several goals set for our project:

- Completed design of major portions of system
- High level of modularity and flexibility, no hard-coding needed in front ends
- Created GUI, web-based, and script-based front ends to the system which used a shared library for actual processing
- Beta-level support for Samba configuration, Alpha-level support for Apache configuration; proved that system can be applied to new configuration objects in a short amount of time.
- Implemented core features, basic configuration editing functional
- Successfully stored configuration data, meta-data, and “type
definitions” in XML

The additional time spent as part of our senior seminar projects allowed us to complete several documentation-related tasks. We added an “Extending Howto” which details the steps needed to create type definition files for new configuration objects. The parsers and the functions and settings which can be over-ridden were fully documented to make extending them easier for developers not familiar with our system. Parsers were written to perform batch conversion of documentation for all directives of both Samba and Apache. This documentation is included in the type definition files and is available “inline” in the front ends.

**Documentation Guide:**

To encourage outside contributions, all of our documentation is kept on our website, http://config4gnu.sf.net. The main documentation is available under the heading “Documentation.” Major sections of the documentation include:

- CFG Article – an article posted on freshmeat.net which explains the purpose of our project (fall semester)
- System Requirements Specification – format description of the features of our project (fall semester)
- Extending HOWTO – information on adding support for additional configuration objects (spring semester)
- Implementation Guide – details on how each major component/feature
is or will be implemented (fall and spring semester)

- Proposed XML Specification – the format used for storing configuration data in XML (fall semester)
- Schematic – a diagram of how the system's components work together (fall semester)

Our website also contains a significant amount of other information including the mailing list and message forums.

Our CD includes the following information:

- /docs - most major documents in PDF format
- /website – an exact copy of our website, containing all documentation, including that in /docs
- /sources – the source code of our project
- /tarballs – compressed versions of /website and /sources
Config4GNU System Requirements Specification

Jason Long

Justin Yackoski

1. Introduction

The Config 4 GNU project is a new open source initiative which will improve the "nightmare" that is the current state of Linux/Unix configuration. We plan to provide a set of standardized tools for users, administrators, and developers which make configuration more efficient and powerful, while remaining true to the Unix philosophies of simplicity and reliability.

1.1. Purpose

The purpose of this System Requirements Specification is to list the various functionality and requirements for the system in a clear and precise way. The System Requirements Specification is to be the guide and vision for the project.

1.2. Scope

The scope of the Config 4 GNU project is dealing with system configuration in all its various flavors:

- Configuration of system settings, such as network interfaces, users and groups, user authentication, and filesystem management.
- Configuration of system applications, such as the Apache webserver, Samba file sharing, DHCP, DNS servers, and mail servers.

Config 4 GNU will work with all sorts of different Linux and Unix-like operating systems. Although the underlying configuration mechanisms will vary on different platforms, the interface presented to the user will not.

The system shall not encompass user-level configuration, i.e. the type of configuration that is stored in a user’s home directory.
1.3. Definitions

In order for maximum clarity in communication of requirements, it is important to define certain terms used in this document.

1.3.1. Application

An Application, also called System Application, in Config4GNU is a program that is typically run by the system administrator(s) and is not customizable per user. This will probably never include a desktop application.

1.3.2. Daemon

A Daemon is an Application that is "always-running" on a system. It typically begins its life at system startup and continues until the machine is shutdown. It may run as the root user or a special user created for the purpose. It does not have a user interface, but waits for other programs to initiate communication or a timed event to happen.

Examples include:

• Apache Web Server
• Cron
• Sendmail Mail Server

1.4. References

• A Solution to the Problem of Configuration in Linux - Jason Long - 28 Sept 2002 - freshmeat.net/articles/view/565/ (http://freshmeat.net/articles/view/565/).
• How to Fix the Unix Configuration Nightmare - Matthew Arnison - 16 Feb 2002 - freshmeat.net/articles/view/400/ (http://freshmeat.net/articles/view/400/).

2. General Description

In this section the proposed system will be described.
2.1. System Goals, Questions, Metrics

For now, this section only contains a broad overview of the major goals. In the future, this section should contain precise procedures for verifying that the goals are being met.

- Provide multiple interfaces (command line, web-based, GUI, LDAP, etc.) that allow users/admins to modify configuration, with relevant documentation (man pages, etc.) integrated into the interface.

- Allow developers to easily create configuration interfaces for their software which will be automatically recognized by our system when the software is installed.

- Standardize the different types of configuration files (INI-style, flat/delimited text, Apache-style, etc.) and provide "parsers" and "unparsers" for all styles.

- Include ways for different operating systems and distributions to tailor the system to their needs.

- Retain the native human-editable config files in /etc as the authoritative copy of system configuration and avoid mangling comments and indentation.

2.2. System Overview

2.2.1. Nodes

A Node in Config4GNU is an aspect of configuration. It is composed of child nodes and properties. If compared to the Windows registry, it is a "key". If compared to a Unix filesystem, it is a "directory."

Nodes belong to a node class, and they have a name that is unique among all the nodes that are contained within the same parent.

2.2.2. Node Class

A Node Class in Config4GNU defines common properties and methods of a configuration node. For properties, it defines possible property names, their default values, their data types, and a description of the property. For methods, they define a name of a method and what actions should be performed when the user wishes to activate that method. In addition, the node class defines methods for creating and deleting a node of the class.

Node classes are subject to inheritance. Node classes can be specified to "extend" another node class, causing all the definitions of the extended class to automatically be present in the sub class.

Node classes shall be identified by a URI (Universal Resource Identifier) and a name. The URI will work like it does in XML namespaces. No meaning is attached to the URI; it merely ensures that third-parties can create their own node classes without the possibility of a naming conflict.


2.2.3. Properties

A Property in Config4GNU is a single configuration setting. It is attached to a configuration node, which may contain many properties. Properties are composed of a name and a value.

2.2.4. Property Value

A Property Value is an abstract type, since it can store many different data types. When possible, a property value will have a way to convert a string to its data type and back.

2.2.5. Data Type

A Data Type is a description of the type of data allowed in a property value. It may also describe the type of widget used to select the value (e.g. a spinner control for a numeric value or a drop-down list for a day of the week).

Many common data types shall be defined by Config4GNU. Third-party applications who use Config4GNU may provide their own data types, in the case that the predefined types do not suffice.

Like node classes, data types will be identified by a URI and a name, ensuring that naming conflicts will not occur if used properly.

2.3. User types and characteristics

2.3.1. System Administrator

The most common user of Config4GNU is the system administrator who wants to make changes to the configuration of an application on the system. This system administrator does not want to edit text files, is not completely familiar with the syntax of the configuration file for the particular application, or wants to somehow automate a configuration file change.

2.3.2. Application Developer

Developers of applications, as defined in this paper, will be interested in Config4GNU supporting their applications. To do this they will need to know how to define the appropriate plugins to let users of Config4GNU use Config4GNU to configure their applications.
2.3.3. Desktop Linux User

Configure will also be designed so that Desktop Linux Users will be able to easily configure their
desktop computer. A difference between the Desktop Linux User and a System Administrator is that
Desktop Linux Users will need to change system properties often and not be concerned very much about
daemons, and the System Administrator will be more interested in working with daemons and other
applications on the system.

2.4. Constraints and assumptions

2.5. External dependencies

2.6. Risks

Risks are those things that could prevent Configure from being a successful project. It is good to
think about what risks there are so developers can concentrate on addressing these issues first. As
development continues, new risks will be discovered, and existing risks will become obsolete. This
section should be updated accordingly.

- A risk is that Configure would be unable to adapt to different versions of applications. For
  instance, what happens when a user upgrades to a version of Apache that has slightly different
  configuration options. The user will want to enter a configuration option for Apache that Configure
doesn’t know about. If Configure is unable to handle this type of situation, it will not be accepted.

- Configure will probably not succeed if major application developers do not decide to support it by
  bundling updated configuration definition files with their distributions.

- Another risk is that the project’s audience are too attached to editing configuration files manually and
  see the project as fixing a problem that does not exist.

- A final risk is that there is some system application that we do not consider that uses a configuration
  format that is not representable with the data model that Configure uses.
3. System Requirements

3.1. Requirements Overview

Requirements are precise and detailed descriptions of various functions and attributes of the proposed system.

3.2. Functional Requirements

Functional requirements are descriptions of specific features or functions of the system. They are things the user can do with the software. They can usually be modeled as having inputs, processing, and outputs.

3.2.1. Create a new configuration node

Describes how a user will create a new configuration "node."

3.2.1.1. Input

- Parent node. The user shall specify the node in which the new node will be created. For example, in a GUI application, the user may select the node by clicking on it once with the mouse. For a command-line utility, it may be specified as an argument.

- Node class. The user shall specify the class of which the new node is a member. In a GUI application, the class may be selected as a submenu of the NEW menu or it may be selected from a dialog box.

- Initial data. Many nodes will require initial data to be created, such as the install location of the configuration file for which it edits, or the name of the object which it is creating. In a GUI application, a form will be presented to the user where the user will enter the necessary information before the object is created.

3.2.1.2. Processing

A configuration class shall define certain actions that need to be performed in order to create a new node of that class. These actions shall be performed using the information provided.

The node, once created, will need to be added to the list of the parent node's children.

3.2.1.3. Output

The output of the creation of the new node will depend on the class, but it will almost certainly require the creation of a new configuration file or addition of data to an existing file. In addition, the data for the
3.2.2. Read a configuration setting

Although Config4GNU does not provide a framework for applications to read their configuration settings, users who want to make changes to the configuration will need to read the existing values to intelligently set new values.

Configuration data is considered a type of data that does not change very often, meaning aggressive caching of values will be allowed when the system becomes distributed.

Reading configuration data shall be subject to access control information.

3.2.2.1. Input

To access a configuration value, the input is a configuration property identifier (i.e. a "path").

3.2.2.2. Processing

No processing is required for this function.

3.2.2.3. Output

The function outputs the value of the specified configuration property.

3.2.2.4. Priority

Mandatory. Required for version 1.0.
3.2.3. Read a nonexistent configuration setting

If the user needs the value of a property that is not actually defined in the configuration file, a "phantom" property should be created, given the default value for the property, and given to the user. This allows forms to contain controls bound to properties that are not necessarily defined in the native configuration file.

Consider this share definition from Samba as an example:

Example 1. A share definition

[myfiles]
  path = /var/myfiles
  writeable = yes

Now consider a form that displays a checkbox for whether or not the share is "browseable." This checkbox should be bound to the property browseable. In order for the checkbox to bind to an actual object, and without requiring the configuration file to contain the implicit "browseable = yes" statement, a phantom property is created and used as the bound target of the checkbox.

See also Changing the content of a Phantom Object.

3.2.4. Changing the content of a Phantom Object

When a phantom object's content is changed from its original default value, it should be saved in the native configuration file the next time the native configuration file is updated. Therefore, as soon as the phantom object is modified, it becomes real.

3.2.5. Notification of added property

When a property is added to an object, notification should be sent to any listeners on that object to the property_added event.

3.2.5.1. Testing

Open two windows containing the same configuration tree. Select an object and make sure that object is visible in both windows. Use one window to add a property. The added property should appear in the other window as well as the first window.
3.2.6. Setting a configuration setting

One of the obvious requirements is for users to be able write values to the configuration files. Setting configuration values not only includes modifying existing values, but adding and removing properties.

Setting configuration data shall be subject to access control information.

3.2.6.1. Input

To write a configuration value, the input is a configuration property identifier (i.e. a "path") and the value to write to that property.

3.2.6.2. Processing

Depending on the application, special actions may need to be performed to "activate" the configuration. For instance, restarting the application will force the program to reread its configuration file.

3.2.6.3. Output

There is no output for this function.

3.2.6.4. Priority

Mandatory. Required for version 1.0.

3.2.7. List all of the properties of a configuration node

Users shall be able to list all of the properties of a configuration node. This way users can see what properties are available for change.

User may list all set properties, as well as unset properties. Unset properties are properties that are defined in the schema file(s) but not actually present in the configuration file. (Most applications do not require every single parameter to be present in a configuration file and define default behavior for parameters that are not present.)

3.2.7.1. Input

To list properties of a configuration node, the input is a configuration node identifier.
3.2.7.2. Processing

No special processing required.

3.2.7.3. Output

The output is a list of configuration property identifiers that are properties of the specified configuration node.

3.2.7.4. Priority

Mandatory. Required for version 1.0.

3.2.8. List all of the child nodes of a configuration node

Since configuration nodes exist within other nodes, users will need to list all of the child nodes of a configuration node.

3.2.8.1. Input

The user shall provide an identifier of the configuration node whose children should be listed.

3.2.8.2. Processing

No special processing required.

3.2.8.3. Output

Config4GNU shall provide the user a list of configuration node identifiers which are children of the specified node.

3.2.8.4. Priority

Mandatory. Required for version 1.0.
3.2.9. Delete a configuration node

From time to time, a user shall have to delete a configuration node.

This action shall be subject to access control lists.

3.2.9.1. Input

The user shall provide Config4GNU with the configuration node identifier of the node that shall be deleted.

3.2.9.2. Processing

The configuration class to which the node belongs shall define the actions needed to delete nodes of that class. These actions shall be performed.

3.2.9.3. Output

Deleting a node will require modification or deletion of the underlying configuration files.

Deleting a node will require removing it from the parent’s list of children.

The user who requested the deletion shall be notified that the node has been deleted.

3.2.9.4. Priority

Mandatory. Required for version 1.0.

3.3. Non-functional Requirements

Non-functional requirements describe characteristics and attributes of the system. They may also describe constraints that must be placed on the system. Examples are performance, efficiency, and security requirements.
3.3.1. Immutability of Configuration Files

When Config4GNU makes modifications to configuration files the original configuration files shall only be modified in the most minimal manner required for the change to take affect.

All non-configuration data shall be preserved, including:
- comments
- whitespace
- order of parameters

3.3.1.1. Testing

A way to test this requirement is to have a bunch of various configuration files, pipe the output of the parsing of each into the unparsers and compare that result with the original, using a utility such as diff.

Example 2. Testing the samba parser/unparser

$ ./parse.ini.pl smb.conf.original ./unparse.ini.pl smb.conf.new
$ diff smb.conf.original smb.conf.new

3.3.1.2. Priority

Mandatory. Required functionality for version 1.0.

3.4. External Interface Requirements

External interface requirements describe the interfaces required to interact with other software and with users. Possible requirements in this area would be the ability to restart an application when its configuration changes and external programs to do user authentication.

3.4.1. Activation of configuration

The configuration definition files shall provide the information necessary for "activating" a configuration—that is, performing whatever actions are necessary to get the application to reread its configuration.
3.4.1.1. Priority

High.

3.4.2. External programs to read/set configuration values

Certain configuration properties may need an external program to set the value. For example, to set a user’s password you may want to invoke the `passwd` program with the necessary arguments rather than directly edit the `passwd` file.

3.4.2.1. Priority

High.

3.4.3. Command-line interface

There shall be a command-line user interface which will allow the basic functionality of Config4GNU to be executed from both interactive and noninteractive shells. This command-line interface will have to allow access to all essential functions described in Functional Requirements.

3.4.3.1. Priority

Mandatory. Required for version 1.0.

3.4.4. Web interface

There shall be a web interface which will allow the basic functionality of Config4GNU to be accessed from other computers via a web browser.

3.4.4.1. Priority

Medium.
3.4.5. Graphical user interface

There shall be a graphical user interface which will allow the functionality of Config4GNU to be accessed from a graphical desktop.

The graphical user interface shall use the Gtk+ toolkit.

3.4.5.1. Priority

Medium.

3.4.6. LDAP interface

There shall be an LDAP interface which will allow the functionality of Config4GNU to be accessed from a common LDAP browser.

To fulfill this requirement, the data model used by Config4GNU must be compatible to the LDAP specifications.

An example LDAP browser is gq, available from biot.com/gq/ (http://biot.com/gq/).

3.4.6.1. Priority

Low.

3.5. Relationships among requirements

4. Quality Attributes
5. Conclusion

5.1. Outstanding Issues
1. Introduction

1.1. Purpose

The purpose of this document is to explain in detail how various parts of the system will be implemented. It is intended for those interested in understanding and/or contributing to the CFG codebase, or those who want to make use of some or all of the CFG system. Some very specific and very abstract details are available in the API Reference (/docs/reference/) and the Schematic Diagram (/docs/schematic.html), respectively.

This document will contain many ideas which previously existed only in the developer’s heads, or perhaps scattered amongst comments in code. Due to its nature every effort will be made to keep this document up-to-date with our currently implementated features as well as any plans for new ones.

This document is not intended to be 100% sane and complete all of the time. It is more important that it stay up to date as much as possible than have complete but incorrect information. Some sections’ implementations are still partly or wholly undetermined, others may not even be included in this information yet.

If you want to comment on this document, please do so via the mailing list. If you have a large change you’d like to suggest we make (or we’ve said we liked your comments and asked you to submit changes to this document), you can send a patch of the source version of this document to the mailing list. The DocBook XML source is available via CVS as /website/docs/implementation/implementation.xml.

1.2. Definitions

In order for maximum clarity in communication of our implementation, it is important to define certain terms used in this document.

1.2.1. Native Configuration File

A native configuration file is a file as stored on a traditional Unix setup and read by a Unix application to
store configuration. Native configuration files vary widely in format from one application to the next, and even from one distribution of Linux to another. The Config4GNU project will maintain existing native configuration file formats, in order to remain compatible to other configuration software and to allow coverage of legacy applications without changes.

2. Dependencies

CFG will be dependent on several technologies and libraries to accomplish various tasks.

2.1. XML

CFG makes heavy use of XML as the common data format between components and between CFG and higher level or third party applications. Several features of XML such as XPath, XSLT, and Schema will be used as necessary by various components.

2.2. Xerces

Xerces (http://xml.apache.org/) is made by the Apache Foundation and provides many advanced XML features, including Schema, XSLT, XPath, etc. It is the leading candidate for use as the primary XML library for CFG. Providing C++, Java 2, and Perl APIs, it shows a lot of promise and could potentially be used by all layers of CFG as the standard XML library.

Use of Xerces will eliminate many of the external Perl modules which are currently required, simplifying several requirements into 1 package.

2.3. Other Dependencies

Other items will be needed for each UI. Most likely, the GUI will be written in GTK, the web version will be written in PHP and/or Perl, so these UIs will of course require additional packages to run. The core, however, should not be dependent on these libraries.

3. Config4GNU Object Model

The Config4GNU object model is a huge virtual XML document. It is called a virtual document because the content of the XML is generated on the fly as needed by "parsing" native configuration files to XML and changes to the XML are automatically implemented in the native configuration files.
3.1. Primitive Objects

Primitive objects just contain some sort of a single value. The value may be a string of any characters, or it may be a number, a date, boolean, or of an enumerated type. Since the underlying format is XML, the value is stored in its string representation.

Here is an example of a primitive object, represented in XML. It contains the date this part of the document was written.

Example 1. A primitive object

<createdate>2002-12-12</createdate>

Various front-ends will provide various widgets to display or modify the content of primitive objects. At the very least, a platform should allow a user to edit the string representation of an object. Since objects inherit from one another, an unrecognized object type can be displayed or edited as if it was a member of the parent type. For instance, if a positive number type was required, but the front-end did not know about positive numbers, it could provide a control for entering any number, since the "positive number" type would extend the "any number" type.

Validation must be provided below the user interface layer. If, as in the prior example, the user interface did not know about "positive numbers", and instead used an "any number" widget, it would be up to the
layer below to reject negative numbers.

The Config4GNU system will define various primitive object types. Widgets will be provided for these types when possible. Some of the types that will be defined by Config4GNU are:

- String
- Number
- Integer
- Date/Time
- Path (i.e. a filename)
- Boolean

Since Config4GNU hopes to use XML Schema in various ways, and the XML Schema specification defines various data types, the data types used by Config4GNU will match with it.

3.2. Normal Objects

Normal objects contain other objects. These child objects are represented in XML as child elements and attributes. These child objects are also called fields or properties. The names of these properties are the element names themselves. The values of these properties are the content of the attributes or child elements and may be normal objects themselves or primitive objects, or complex objects.

Any whitespace that is found between child elements of a normal object is ignored. Any text found between child elements is considered invalid, and should be treated the same as a parsing error.

Multi-valued properties are allowed using this model. Just have a child element with the same name occur more than once and you have multiple values.

This model has many of the features we want: it's simple, flexible, easily allows multi-valued attributes, and is extensible--all you need to do is add elements or attributes.

In the following example, an XML representation of a runlevel configuration object--which is a normal object, we see some of these concepts. The default-runlevel child element is a simple text property of the runlevel configuration. There are multiple service child elements, these are themselves normal objects and there can be many of them in the runlevel configuration. The service element has two properties that are primitive objects: name and runlevels.

Example 2. A normal object

```
<runlevels-config>
  <default-runlevel>default</default-runlevel>
  <service>
    <name>bootmisc</name>
  </service>
</runlevels-config>
```
<runlevels>boot</runlevels>
</service>
<service>
  <name>samba</name>
  <runlevels>default</runlevels>
</service>
</runlevels-config>

Normal objects can also have methods. These are actions that can be performed on an object. Since methods are not per-instance but rather per-class, they are not defined in the XML data files but in the XML type (or schema) definition files.

3.3. Complex Objects

Some XML elements contain mixed text and elements. A great example would be in XHTML, where a string of text might contain an img element to insert an image in the middle of some text. These objects are considered complex, because they are more complicated to handle. As with other elements, widgets can be defined to handle them. In the fallback case, the user will have to edit the raw XML.

3.4. Object Names

Very simply, the "name" of an object is the value of its name property. This may be specified directly in the instance data, using the <name> tag, by a default value in the Schema file, or by programmatically overriding the "get" method for the "name" property.

The name of an object is used when displaying object among other objects in the object tree (see Object Tree). There may be other uses as well.

3.5. Object Children

The children of an object are a subset of an object’s properties. Children are those properties that should be treated as separate objects from the parent object but would be considered "contained" by the parent object. In the GUI, child objects appear next to an object in the configuration tree that appears in the left pane.

There are two ways to specify what the children of an object are. Remember that children objects are properties as well, because the children of an object are a subset of the properties of an object. To identify children, a flag must be added to the property definition to mark it as a child object.
The first type of child object is a property that acts as a separate container for an object's children. A common way to use this is with a "children" property. Any child objects of the "children" property are child objects of this object. Here is an example XML representation that uses the children element.

**Example 3. Identifying children using the children property**

```xml
<config>
  <children>
    <sambashare>
      <name>public</name>
      ...
    </sambashare>
    <sambashare>
      <name>ProjectFiles</name>
      ...
    </sambashare>
    <sambaprinter>
      <name>Office Printer</name>
      ...
    </sambaprinter>
  </children>
</config>
```

This XML representation would generate the following tree representation:

- Samba configuration
  - public
  - ProjectFiles
  - Office Printer

In the current version of Config4GNU, any property named children behaves this way. In the future, there will be a flag that you can add to the property definition of children which will make it behave this way.

The other type of child object is a property that becomes a child itself. Take this example:

**Example 4. Properties as Children**

```xml
<runlevels-config>
  <runlevels>
    <runlevel>
      <name>2</name>
      <description>No network</description>
    </runlevel>
    <runlevel>
      <name>3</name>
      <description>Default</description>
    </runlevel>
  </runlevels>
</runlevels-config>
```
</runlevels>
<services>
  <service>
    <name>named</name>
    <runlevels>3</runlevels>
  </service>
</services>
</runlevels-config>

The children of the runlevels-config object should be "Runlevels" and "Services". The following list shows how it would appear in a tree display. Assume that we defined that the children of runlevels-conf should be Runlevels and Services, and that we applied names to elements that do not have name elements.

- Runlevel Configuration
  - Runlevels
    - 2
    - 3
  - Services
    - named

In the current version of Config4GNU, identify this type of child object with an attribute child="true" in the property definition for the property that should become a child.

3.6. Links

Links are connections between two objects in the configuration system. Whenever an object references something that is another object in the Config4GNU system, it should identify it in the way that clients can connect it to that object.

For example, if you are representing a filesystem, and you want to identify the owner of the file, identify the owner of the file not by specifying the username, but by identifying the object that represents that user. For more information on identifying objects, see Identifying Objects below.

This may be complicated at times, particularly if the two objects are located in separate documents. In this case, it may be acceptable to use a less detailed method to identify the object. However, when two objects are in the same document, they should always refer to each other by object not by contents.
3.7. Identifying Objects

The Config4GNU system must define a way to globally identify a specific aspect of configuration. This is important, for example, to identify associations between different aspects of configuration (e.g. access control lists).

An absolute path would have to include a mechanism to globally identify a computer, perhaps using a fully qualified domain name or an IP address, and then the specific configuration item on the machine. However, because specific instances of configuration should not be tailored to a specific machine but should be transferrable to different environments, several shortcuts should be available:

**FRAGMENT_ROOT**

Identifies root of a particular aspect of configuration. For example, the FRAGMENT_ROOT inside a Samba configuration file would point to the root of the Samba configuration file.

**LOCAL_COMPUTER_ROOT**

Identifies root of configuration for the local computer in which the context node exists

**ORGANIZATION_ROOT**

Identifies a local root of a configuration hierarchy including all the computers of a particular organization

**GLOBAL_ROOT**

Identifies the real root of the virtual XML document which would conceptually include all organizations in the world

4. The User Interface Layer

The user interface layer consists of the various front-ends that allow users to read and manipulate configuration data. Front-ends will be available on a variety of platforms, including Gtk+/Gnome, Qt/KDE, Curses, command-line, Web interface, and programming interfaces such as CORBA and LDAP.
4.1. Overview

The user interface is a specific UI (GUI, Web, Command-line, etc.) which calls library functions to perform any real work to read and manipulate XML files. This common library will be used by all UIs to perform XML editing tasks. The UIs will be generic, extendible tree and form based XML editors which will happen to have lots of configuration-XML editing extensions supplied with it. The library functions needed to perform XML tasks not specific to configuration will be included with the generic XML library, separate from the main CFG library so that it will be possible to use the UIs for editing other XML documents without loading the main CFG library.

The UI loads one XML document via library calls. This document may contain links to other XML documents. These links will allow entire documents to be logically attached to XML nodes of the parent document to facilitate editing.

The actual UI itself will contain no configuration specific code, nor will it contain code to directly manipulate XML. All this will be handled by library calls so multiple UIs can easily be built.
4.2. Root Node

When an XML representation of configuration is loaded into the client, it will be loaded from some base, called the Root Node. This root node could be the root of just one aspect of configuration, such as Samba if you only want to edit Samba's configuration, or more commonly it would be the root node of the local computer, showing all the configuration available on the computer. A third case for a root node would be a node representing the base for all the computers in an organization.

To change whatever the current root node is, look for an "open" feature in the front-end. This "open" command will allow you to specify what aspect of configuration you want to load, and that aspect's root will become the current root.

The current root will be displayed as the root of the "object tree."

4.3. Object Tree

Many user interfaces will support the metaphor of an "object tree," to allow easy access to the various sections of a configuration file. An example of the object tree is shown in the left pane of the above screenshot (the Gtk+ client).

The object tree is constructed by starting with whatever the current "root" node is. Any element of any XML document may be a root node, but it will most commonly be the root node of some aspect of configuration, a node representing all the configuration available on a single computer, or a node representing all the configuration available on all the computers in an organization.

For every element that appears in the object tree (starting with the root node), its name will be displayed. The name of an object is defined by the object type, but is usually simply the content of the name property (see Object Names).

Also for every element appearing in the object tree, one or more children may appear. The children of an element can be defined in multiple ways depending on what type of node it is (see Object Children). In this manner, many of the "objects" found in an XML document can be selected.

4.4. Current Node

Using the object tree, a user can select a node to edit by clicking on the respective item in the object tree. When an object is selected, forms appropriate for editing the object are loaded in the right pane.
4.5. Forms

Forms give users a window filled with widgets and labels of various types where users can see the important properties of an object and edit it using text boxes, list boxes, buttons, and other common controls. The information is presented in an organized way and may be split into multiple tabs or pages if there is a lot of data to present.

Forms are defined using an implementation-independent language, such as XForms. The same form definition files could be used to create Gtk+ dialog boxes and HTML web pages. Forms for some aspect of configuration are distributed with the program that they are meant to configure, by Config4GNU itself, or some third-party. As new versions of a particular distribution or system application are released, the forms may need to be updated to support new features.

4.6. Property List Form

The property list is a default form that is available for all "normal objects". It is a form that displays a list of all properties and their values. The list will have two columns. In the first column are the names of the properties, and in the second column are the respective values. When you click on a value (assuming you have permission to edit it), depending on the type of data it contains, the cell will be replaced by an inline editable widget, or another screen will appear allowing you to edit the property.

4.7. Widgets

Primitive objects will typically have a simple widget defined to display its value and allow the user to manipulate it. For example, an element which specifies a path may have a Browse button next to it which allows the user to select a directory on their file system to use as the value. In an SVG document, perhaps a color element would have a color wheel widget to allow the user to select a RGB color value and see a sample of it in real time. This is done in the XML Schema by defining precisely what data type an element should have. Then the front-ends use their knowledge of the various data types to select the appropriate widget.

4.8. Validation

Validation on the user interface level is the first stage of validation when a user uses Config4GNU to make a change to configuration. Validation on this level is validation on a "widget" level. For example, a widget for entering a date will ensure that the value entered is in a date format and corresponds to an actual date.

Validation will continue in the lower layers as the user-initiated change in configuration trickles down to those layers.
The user interface layer must also be ready to handle data already in the configuration file that does not appear to be valid. Continuing the example above, the date found in the native configuration file is not a valid date according to the widget used to display/edit the date in the user interface. In this case, the user interface should probably accommodate by using a less restrictive widget for displaying/gathering the data (e.g. a simple text entry).

4.9. Shared XML Library

The library shared by all UIs to perform tasks not restricted solely to each UI (such as handling mouse clicks in a GUI, generating HTML in a web-based version, etc). The UIs will process the user input and handle the rendering of the UI, and then use functions in this library to do the real work.

5. CFG Upper Layer

The upper layer of CFG provides an interface to the CFG system. Unless circumvented manually, it handles ALL configuration specific actions required to load and save a virtual CFG XML document.

5.1. Authorization & Authentication

Before accessing the configuration data, a client must select an Authority. The selected authority will determine what level of access the user will have to the configuration. How the client obtains an authority depends on what type of application it is. If it is a normal application, running as the user, the default authority will be an object representing the current user. If the client is running as a system user, e.g. apache, it will want to obtain an authority representing the user that is using the program.

CfgAuthority::get_initial_authority() will return an authority representing the user ID of the current process. A program that the user starts will want to use this method because the user has already authenticated.

CfgAuthority::get_local_user(username) will return an authority representing the user specified with the username argument. Programs that are running as a system user will use this method after obtaining a username from the user. Once receiving the authority object, the client will need to authenticate to make it valid.

Once an authority object is obtained, it may need to be authenticated. Simply call the authenticate method, passing it the user's password (or whatever credentials required by the authority).

Finally, when an authenticated authority is in possession, the client can use the get_configuration_root method to get a reference to the configuration tree.
5.2. Configuration Interface

The Configuration Interface provides the API for accessing and modifying configuration data. The following several sections will describe the various classes and functions that are available to them.

5.2.1. CfgObject

CfgObject is a class that provides an interface to a node in the configuration tree. It can be considered a simple wrapper to an XML Node, but it provides several features not found in the XML DOM API. These features include functions particular to the configuration domain of Config4GNU. These features are:

- Signals for when the structure and contents of the object are changed.
- Access to object and property type information.
- Automatic loading and saving of XML as needed.
- Dynamic merging of multiple XML documents to present a single hierarchy of configuration.

Signals allow parts of a program to register themselves as listeners for certain events, such as "value changed" or "property added". Whenever the CfgObject class is used to make a change to the object, it will automatically "emit" the appropriate signals, and this way any listeners to that event will be notified. An example of this is a text entry control that automatically updates its contents whenever the object it is bound to is updated.

The CfgObject also automatically handles objects that import other configuration documents. For example, consider the following fragment of config4gnu.xml.

Example 6. An Import Object

```xml
<application>
  <name>Samba Configuration</name>
  <module>CFGXML::Parser::App::Samba::Common::SambaParser</module>
  <configfile>/etc/samba/smb.conf</configfile>
</application>
```

The object represented by the application tag here is an "import" object. When double-clicked in the GUI, the Config4GNU system will run the parser specified by the module tag using the file specified by the configfile tag and the XML representation generated by the parser will be inserted as if it were in the application element.
6. CFG Middle Layer

The middle layer manages the loading and saving of configuration XML documents. It is responsible for authorizing a user to perform reads and writes to a particular configuration object. It handles a large amount of the system’s validation. Because it does the loading and saving, it may run as a separate, privilege process (one that has full access to the native configuration files). If it runs as a separate process, communication between the upper layer and this layer will use some sort of IPC. Preferably, this IPC would be flexible and extendible, like CORBA or SOAP. Whatever technology is chosen, however, security of the privileged process is absolutely and totally required.

6.1. Loading and Saving

The front-ends will have various mechanisms through which an XML representation of a configuration can be loaded or saved. To load a part of configuration, an "import" object would be created, containing properties such as "parser-module" and "configfile" that contain the name of the parser to run and the location of the exact configuration to load. Once this import object is created, the client will invoke its "expand" method. The "expand" method runs the requested parser and makes the object tree generated by the parser available as properties and children of the import object itself.

To save a part of configuration, one would use the "save" method of the import object. This save method again runs the appropriate parser, provides it the updated object tree, and has it rewrite the native configuration file.

As part of the load/save process, attention will be paid to access control, logging changes, etc. in the lower layers. In some cases, there may not actually be "native configuration files," in which the XML will simply be saved as-is to a file on disk.

6.2. File Access and Caching

All file access related to parsers will be handled by the section of the middle layer which interfaces with the bottom layer. This means that for cache hits, the parsers will not be consulted at all, and provides a point to track which files are read/written and make them available when needed.

6.2.1. Parser Access to Files

Parsers need access to various config file on multiple machines. To handle reading/writing to these files, the parsers must go through the middle layer’s API. Initially, all parsers will be given access to a single main file, which the middle layer will be responsible for opening either at the start or end of the parser’s execution (depending on whether loading or saving is taking place). Subsequent files must be requested by the parsers through the middle layer’s API. For loading, the file will be given to the parser as an array containing each line of the file. For saving, the parser will give the middle layer a string to write out.
To help uniquely identify files, any file other than the main config file must be specified by its full path by the parser during loading/saving API calls. For the primary config file, the filename should be left blank in the API calls. The full path to the config file must be calculated by the parsers, as the way of determining the full path varies between config file formats. At this time there is no known config file format which specifies an included config file which is on a different host than the primary config file, so the host for included files will be assumed by the middle layer to be the same as the primary config file.

The following functions will be used for parser/middle layer interaction:

```c
string[] load_primary_file(void);
Returns an array containing each line of the primary config file. The parser does not know
primary config file, but the middle layer should be able to easily determine it. If the pri-
mary config file is a directory, the array should be a list of files in that di-
rectory. If there is no primary config file, returns null.
```

```c
void save_primary_file(string $output - The string to save);
Saves $output into the primary config file.
```

```c
string[] load_secondary_file(string $fullpath - The file to load);
Returns an array containing each line of the specified config file. If $full-
path is a directory, the array should be a list of files in that directory. Re-
turns null on failure. Note that the file may not really be located at $full-
path, $fullpath is just where the parser thinks the file should be. The middle layer may l-
other host, add a prefix to the path, etc.
```

```c
void save_secondary_file(string $output - The string to save, string
$fullpath - The file to save to);
Saves $output into the secondary config file $fullpath. Note that the file may not re-
ally be located at $fullpath, $fullpath is just where the parser thinks the file should go.
```

During loading, the middle layer is responsible for appending XML to the root node to track files which
the XML document depends on and maintaining the most recently modified file's timestamp for
inclusion into the XML.

### 6.2.2. Storage of Cached XML & Data

The XML will be stored in a directory to be determined (probably .cfg in the user's home directory until
privilege escalation code is added to CFG) with the same permissions as the original primary config file.
Ideally, a single system-wide location would be used. A subdirectory will be created and named after
each host for which XML data is cached. The domain name will be used if available, otherwise
"localhost" or the primary IP of the machine will be used. Slashes (forward or backward) in the full path of the file will be replaced with underscores (double slashes will be replaced by a single underscore), and any underscores already in the path will be replaced with double underscores. Additionally, the unix timestamp of the modification time of the primary file will be appended to the filename. So that the file /etc/weird_app/main.conf for localhost modified at timestamp 500000000 will be stored in CACHEDIR/localhost/etc_weird__app_main.conf.500000000. Note that if main.conf were to include a file with a later modification time, the latest modification time of all the relevant files would be appended instead.

The file will be owned by the user running CFG, or by root when system-wide caching is implemented, and only be readable or writable by the owner (600).

Inside the actual XML, included config files will be listed directly under the root XML node. For example, if a main Samba configuration file /etc/samba/smb.conf includes the files /etc/samba/smb.extra.conf and /etc/samba/smb.more.conf, then the following XML would be stored in the cache, assuming the most recently modified time of the 3 files is 5000000.

Example 7. XML cache file etc_samba_smb.conf.5000000

```
<config>
  <include primary="true" host="localhost" owner="root"
      group="root" mode="0644">
    /etc/samba/smb.conf
  </include>
  <include owner="root" group="root" mode="0644">
    /etc/samba/smb.extra.conf
  </include>
  <include owner="bob" group="users" mode="0600">
    /etc/samba/smb.more.conf
  </include>
  <dependstamp>5000000</dependstamp>
  <include owner="root" group="root" mode="0644">
    /etc/samba/smb.conf
  </include>
  ...
</config>
```

Only 1 version of each config file is kept in the main directory for each host. Before a new cache file is written, older versions must be moved to an attic subdirectory for each host. This ensures that when checking the cache, the library will not have to waste time locating the most recent file. If a crash occurs, the worst case will be that the cache file will be prematurely moved to the attic subdirectory.

6.2.3. Caching

When an XML document load is requested, the cache is checked to see if a matching cache file for the given host and full path exists. Only one version of each config file for a specific host will ever be present in the main subdirectory for the host. If such a file is found, the timestamp appended to it must be equal
to or greater than the timestamp of the primary config file. If so, each non-primary dependfile's timestamp must not be higher than the dependstamp. If so, the cache file is used.

If one or more of those conditions are not met (a cache miss), then the parser is run to generate the XML document being requested. The library appends dependfile and dependstamp tags to the XML as described above, and the file is stored in the cache. Any previous versions of the config file are moved into an attic subdirectory specific to the appropriate host.

During saving, the XML document is saved to the cache immediately after the config files are finished being written by the parser and middle layer. Any previous versions of the config file are moved into an attic subdirectory specific to the appropriate host.

6.3. Translations

Several types of translations will be done. In some cases, two XML documents it requested from two different parsers will need to be merged. For example, it may request an XML representation of a config file from the parser for that config file format, and then request an XML representation of the relevant man page from a man page parser. To make displaying context-sensitive help easier for the UI, it will then merge the data from the man page into the relevant nodes in the XML for the actual config data.

In some cases, default values will need to be added to the XML on its way to the UI, or be stripped from the XML on its way back to the unparsers. What cannot be done in a compatible way by the Schema files (default value adding) will be done here.

6.4. Plugins

Plugin capability will be available for the middle layer. Extended features such as more advanced authentication, remote configuration editing, logging and rollback, etc. will be available through plugins. These plugins will be non-specific to the application being configured or data in the XML. It will simply perform an advanced task with the XML.

6.5. Access Control

There are two aspects of access control. First, there is the process of authenticating the caller as who it says it is. Second, there is the process of authorizing the caller with given abilities.

6.5.1. Authentication

Users may select with what identities they will connect to the Config4GNU system. The identity will be
some node that exists in the Config4GNU object model. Initially, the only identity possible to connect as will be the current user. Since users and groups will exist in the Config4GNU object model for each machine, the authenticated identity will be the user object for the current user.

6.5.2. Authorization

Authorization is done on the target objects. That is, to grant a user access to change a certain configuration, I will go to that certain configuration and add the user to the list of things allowed to make changes. This list is called an access control list (ACL) because it is a list of access control items.

Since the Config4GNU object model is XML-based, the following elements will be defined to represent access-control information.

**Access Control Elements**

access-control
   specifies an inline access control list

access-class
   references one or more external access control list definitions

The access-control allows an ACL to be specified directly with the relevant node that it covers (the context node). The ACL will be a list of users and groups and their associated privileges. The order is important, for the first access control item where the subject (i.e. user or group) matches will be the one that determines access.

6.5.3. Examples

Example 8. Use of the access-control element

```xml
<access-control>
   <aci>
      <match>user[name=root]</match>
      <privilege>readwrite</privilege>
   </aci>
   <aci>
      <match>user[name=jdoe]</match>
      <privilege>readonly</privilege>
   </aci>
   <aci>
      <match>group[name=admins]</match>
      <privilege>readwrite</privilege>
   </aci>
   <default>
      <privilege>none</privilege>
   </default>
</access-control>
```
In the above example, assume that jdoe is a member of the admins group. In this case, jdoe will have read-only access, despite the fact that the admins group has readwrite access. This is because jdoe occurs first and therefore will be matched.

**Example 9. Use of the access-class element**

```xml
<access-class>admin</access-class>
```

This effectively "includes" all the access control items that are defined in the access control class named "admin". The location where these external access control classes are defined is not yet determined.

### 6.5.4. Definitions

**Access Control Definitions**

access control list

zero or more access control items that have the same context node

access control item

defines a subject and a privilege for an context node

context node

the XML node on which an access control list is attached and for which it defines access

privilege

one of none, readonly, or readwrite

subject

any Config4GNU object, typically a user or a user group. If a "container" object is specified, then all objects that exist within the container will implicitly be given the same privilege

### 6.6. Query Subsystem

This subsystem will be responsible for all queries to the system. It should incorporate any optimizations that it can, including for instance, heavy caching of results.

An example of a query would be: "list all user accounts on all servers where the name of the user is jdoe." Obviously this kind of query might take some time if the computer this is executed on doesn't already have some idea of what users already exist.
7. CFG Bottom Layer

The bottom layer handles actual manipulation of the real configuration data. The parsers & unparsers read data on standard in and write on standard out, so they need not know the actual path to the source or destination. This allows the middle layer the possibility to transparently handle remote configuration editing without the parsers needing to know.

7.1. Parsers

There are two types of parsers: primary parsers and secondary parsers. Primary parsers do a direct translation between one native configuration file to an equivalent XML representation. Secondary parsers operate a level above primary parsers, using knowledge about a specific distribution to pull information from one or more configuration files to create an XML representation that is not distribution-specific.

7.1.1. Primary Parsers

Primary parsers do a simple translation from a native configuration file format to an equivalent XML representation. It may tailor the XML output to the specific application it is working for, and it may apply knowledge of what data types are allowed where, but the structure of the XML representation is equivalent to the native file and all information in the native file is represented in the XML representation (including whitespace and comments).

7.1.2. Secondary Parsers

Secondary parsers, which really should not be called parsers at all, are used to work with an XML representation of idea of configuration, not a specific configuration file. An example of this might be an XML representation of network settings, which might pull information from /etc/hosts, /etc/resolv.conf and several distribution specific files that tell how the network interfaces are setup.

Perhaps a better example is an XML representation of all shares that are exported on a computer. It would include shares provided by NFS, Samba, anonymous FTP, and a web server. Needless to say, there is no single configuration file that can provide this information. When the user asks for this XML representation, information would need to be pulled from at least four different configuration files.

7.2. Writing Parsers

The parsers are written in Perl and all (un)parsers are implemented by extending the CFGXML::Parser module, which provides basic parsing functions. Parsers can further be extended by adding rewrite rules
and changing other minor behavior without having to re-write a new parser from scratch. For example, the Samba parser extends the Ini parser by adding various rewrite rules specific to Samba configuration.

The parsers are written in Perl (as opposed to C/C++) because this allows new Perl modules to easily be dropped in as part of the install of another package. For example, if Apache was not natively supported by CFG (an unlikely case), the Apache maintainers could include a Perl module which extended the $CFGXML::Parser module. This allows instant support for new applications.

7.2.1. Extending

Parsers must be carefully written to make extending possible. For example, the delimited file parser is defined to use the $self->{DELIMITER} variable, which can be redefined in new modules in init(). In cases where an existing parser is not sufficiently flexible, it should be re-written to become more flexible while maintaining its default behavior. Modified behavior can then be achieved by setting additional variables in init() in a new module which extends the enhanced one.

To add support for an additional file format, first create a parser for the generic file format. Then, extend that parser by setting more specific options. For example, you can create an Apache-style format parser, then extend it to support Apache files. This is desirable because then someone else can easily extend your Apache-style parser to support Proftpd files, since its basic format is extremely similar.

7.2.2. Rewriting tags

The $CFGXML::Parser module supports the use of rewrite rules. They can be used by setting several variables which are clarified in src/parsers/README. Parsers for generic formats (INI, Delimited, etc.) should contain little or no rewrite rules.

7.3. Distribution Handling

When distributions vary in the way they configure various things, such as runlevels and network settings, separate parsers will exist for each distribution. The parsers will be different from each other, but their output will follow the same format, and in this way the end-user will not need to know the details of a particular distribution.

The only time a user will need to be concerned with the details of a particular distribution is when incorporating a new aspect of configuration. This is because when you add a new configuration, you must select what parser to run to get that configuration. (Remember that parsers will vary from one distribution to another.) However, once the parser is selected, the configuration can be manipulated without knowing what file format it is, what files it uses, and what parsers are being run.
For example, you have a GUI client of Config4GNU running and you want to use it to configure your system's runlevels. You select the File menu and choose New Configuration. You will see a dialog with a list of parsers. In this dialog, you will find a group of parsers for your distribution (e.g. Gentoo) and in this group you find a Runlevels parser. Select this parser and you will see the runlevel configuration appear in the client. At this point you can configure runlevels just like runlevels on any distribution.

If the new configuration was created inside another document, the parser you select, as well as any parameters you enter to customize the behavior of the parser, will be saved inside this "parent" document. In this way, you really do not need to remember the details of your distribution. In fact, if each distribution provides a configuration document that contains entries for every node of configuration and what parser to use for each, the end-user never has to manipulate new configurations at all.

You may wonder why we are so concerned about what distribution a user is using, and why knowing the differences between distributions is so important. Distributions tend to configure various features uniquely and modify application behaviors to suit their customers best. One of the most prevalent things distributions do to customize software is change the location of their configuration files—a very important factor when considering configuration software, such as Config4GNU.

8. External Entities

This section describes how the Config4GNU system will interact with external entities. These are things that Config4GNU cannot necessarily control, being outside the boundaries of what constitutes a running instance of Config4GNU.

8.1. Native Config Files

The native config files are maintained as the authoritative version of configuration. When a UI requests an XML representation of the current configuration, the native files will be checked to ensure they have not been hand-edited since the last XML cache of each file was made (during the previous usage of the program), and either the cached XML or freshly generated XML is used depending on the situation.

8.2. Remote Computers

The Config4GNU system will be able to read and write configuration on remote computers. This remote access will be done by requesting the remote computer to generate an XML representation of its configuration (by invoking a command-line CFG over ssh). Thus Config4GNU will need to be installed on all computers that you wish to configure using the system.

All communication with remote computers should use a secure protocol, particularly since some configuration data contains sensitive data. However, even if the data is not sensitive, with the availability
of free encryption utilities, there is no reason not to make communication secure.

9. Supporting Files

Some files, such as Schemas, files which contain mapping information, distro-specific information, etc. are used by more than one layer and thus are mentioned here.

9.1. Parser Definition Files

Remember that parsers are the "bottom layer" programs that can generate XML representations of configuration data and write or modify configuration files from the XML representations.

Parser definition files map a configuration identifier, such as "samba", to the information needed to translate between underlying configuration data and an XML representation of the configuration data. This information includes:

- translated names and descriptions of the parser
- what back-end parser to run
- what parameters the parser understands (e.g. location of configuration file, version of application), their descriptions, and default values
- how to communicate the values of parameters to the parser

9.2. Schema Files

Schema files define the structure of XML configuration documents. They define what elements are allowed where, and what the content of each element is allowed to be. Elements can be defined as "simple" or "complex." Simple elements contain text content only; examples might include String, Number, Date, or Path. Complex elements can contain child elements and/or text content. In this case the schema would list what child elements are expected.

Schema files declare a type for all elements it defines. Some types (those that are considered fundamental) are defined by the XML Schema specification or by Config4GNU. Other types are defined by the XML Schema file they are referenced in.
Index
1. Introduction

1.1. Purpose

The purpose of this document is to explain in detail how the CFG system can be extended to configure new things without having to modify existing CFG code. Our modular architecture lets you do practically anything you need to do while at the same time performing common or complicated tasks for you by either extending an existing configuration entity or by using our libraries.

If you want to comment on this document, please do so via the mailing list. If you have a large change you'd like to suggest we make (or we've said we liked your comments and asked you to submit changes to this document), you can send a patch of the source of this document to the mailing list. The DocBook XML source is available via CVS as /website/docs/extending/extending.xml.

2. Frontend

This section is not yet finished, please email us if you are interested in contributing to the front ends.

3. Backend

The backend consists primarily of parsers which are responsible for the translation between XML and the native config files. The parsers may be written in any language, however it is strongly encouraged to use Perl because nearly all *nix systems have Perl, and a useful Perl module exists which handles many common tasks in a uniform way. To add support for a new configuration entity, an existing parser can often be extended with minimal effort, leaving only the task of creating type definitions for your parser's XML.

Note: When referring to parsers, often the CFGXML::Parser:: beginning of the parser's full module name is omitted for readability.
3.1. Extending the Parser Backend

The existing CFGXML::Parser, config4gnu::CfgObject, and related Perl modules provide many functions and options which handle most of the work for you. For reference, if your parser is more than 200 lines at most then you're probably either not using the libraries to the extent you could be, or you should put some of your code into functions and submit them for inclusion in the standard CFGXML::Parser module.

To create a new parser, one of two approaches can be used. Either subclass an existing Perl parser using Perl's inheritance system or write a completely new parser from scratch. Generally, you should use the first approach if a parser for the configuration file format that you're adding already exists, for example the App::Samba::Default parser was created by extending the Common::Ini parser which handles the INI-style formatted configuration files.

If no parser for the format you need exists create two parsers, the generic parser for the format (i.e., INI), and the parser for the particular application which you are adding support for. Generally the generic parser goes in CFGXML::Parser::Common::YourFormat and the specific parser goes in either CFGXML::Parser::App or CFGXML::Parser::Distro. Doing this takes little extra work but allows others to extend your base parser for other applications.

3.2. CFGXML::Parser Module

The CFGXML::Parser is the base module (aka class) for all other parsers. Even parsers you write from scratch will benefit from extending this module.

3.2.1. Options

The following options are available and should be set in your parser's init() function if you want to use something other than the default value.

Example 1. Setting a parser option

```perl
sub init
{
  my $self = shift;       #req'd for perl's object-orientedness
  #call parent class's init first so our settings aren't over-ridden
  $self->SUPER::init();
  #set root XML tag to <apache-config>
  $self->{ROOTTAG} = 'apache-config';
}
```

Note: If you add your own option to your parser, be sure to use the name listed below if your option is very similar or identical.
• BOOLEANRULES - A hash containing mappings from true to false for boolean type values. The keys of the hash are the allowed true values, while the values in the hash are the corresponding false value for each true value. By default, it recognizes: true/false, yes/no, 1/0, True/False, Yes/No, enabled/disabled, Enabled/Disabled

• CLOSECOMMENTALLOWED - For Common::Apache, whether close comments (those on the same line as the closing tag of a section) are permitted. Default is 1.

• COMMENTCHAR - The single character used to detect the start of a comment in CFGXML::Parser's isComment() and tokenize() functions. Default is # (pound symbol).

• COMMENTCHAR2 - The secondary single character used to detect the start of a comment in CFGXML::Parser's isComment() and tokenize() functions in addition to the COMMENTCHAR value. Default is disabled (empty string).

• DEBUG - A flag used to output extra debugging information from various functions/parsers. Normally this is only used during parser testing, as often the extra output is not valid XML or native config file. Default is false.

• DELIMITEDEXTRAFIELDS - For Common::Delimited, whether extra fields are allowed. If 1 (true), any fields encountered after all of the fields found in DELIMITEDFIELDTAGS, any remaining text is added in an 'extra' tag, and delimiters in the extra field are ignored. If 0 (false), an error will occur if any additional fields are encountered which are not found in DELIMITEDFIELDTAGS. Default is 1 (true).

• DELIMITEDFIELDTAGS - For Common::Delimited, the list of XML tags to use for each major field in the delimited file. If empty, PROPERTYRULESDEFAULDT is used instead. If you specify these tags, then during saving, the XML need not be in a particular order for the config file to be constructed properly.

• DELIMITER - For Common::Delimited, the main delimiter used to split a section (line) into its properties. Default is the space character.

• PROPERTYINVERTRULES - A hash containing arrays listing boolean antonyms. The keys for the hash are the names of the tag used to store the antonyms, and the elements in each array are the antonyms to be converted. The name and value of the property are preserved in most cases, but are displayed in the UIs as their opposite. Note: All antonym must be listed for this feature to work. Note also that this feature is not fully implemented or working yet. Note that the property's name is converted to lowercase and spaces are replaced with underscores before comparison takes place. Default is empty.

• PROPERTYRULES - A hash containing arrays, where the keys of the hash are the target XML tag names, and the arrays containing the criterion to convert if the property's name matches. The original name of the configuration directive is preserved when the config file is saved. Note that the property’s name is converted to lowercase and spaces are replaced with underscores before comparison takes place. Default is empty.

• PROPERTYRULESDEFAULDT - The default tag to use for properties, if no tag is specified and one cannot be determined automatically. Default is property.
RIGHTCOMMENTALLOWED - For some parsers, whether right comments (those to the right of configuration/section data as opposed to on a line by themselves) are allowed. Default is 1.

ROOTTAG - The XML tag to use for the root of the XML generated by the parser. You most likely want to set this to something unique, such as samba-config, ini-style, etc. You then need a matching file in data/classes/ROOTTAG.xml. Generally, parsers for the same thing but on different distros or for different versions of things should have the same ROOTTAG (since they should use the same XML when possible). Default is generic-config.

RUNLEVELDIR - This directive has two possible contexts, but is always the path to the directory tree containing symlinks for services enabled on each runlevel, and the default is /etc/runlevels. For Common::SysVRunlevel the default is /etc/rc.d, and for Distro::Gentoo::Runlevel the default is /etc/runlevels.

RUNLEVELS - For Common::SysVRunlevel, an array containing the list of possible runlevels. Default is 0, 1, 2, 3, 4, 5, 6.

SECONDARYDELIMITER - Common::Delimited, the secondary delimiter to use to split properties into sub-values. No quoting of the SECONDARYDELIMITER is supported. Default is , (comma). Note that this is ignored if SECONDARYDELIMITERLIST is anything other than an empty array.

SECONDARYDELIMITERLIST - Common::Delimited, the secondary delimiters to use to split each field into sub-values. To use, create an array with one entry (in order) for each field in your delimited field. Each item in the array is the delimiter which gets used in the split to create the sub-values. By default, this array is empty and is ignored, instead SECONDARYDELIMITER is used for all fields. Note that if you put anything in this list, SECONDARYDELIMITER is ignored for all fields.

SECTIONRULES - A hash containing arrays, where the keys of the hash are the target XML tag names, and the arrays containing the criterion to convert if the section's name matches. The original name of the configuration section is preserved when the config file is saved. Note that the section's name is converted to lowercase and spaces are replaced with underscores before comparison takes place. Default is empty.

SECTIONRULESDEFAULT - The default tag to use for sections, if no tag is specified and one cannot be determined automatically. Default is section.

SERVICESDIR - This option has possible contexts, but is always the directory containing all available services. For Common::SysVRunlevel the default is /etc/rc.d/init.d. For Distro::Gentoo::Runlevel the default is /etc/init.d.

VALUETAGS - A hash containing arrays, where the keys of the hash are the XML tag name of the parent, and the array contains the custom value tags to be used for that parent's value tags. See addValueArray for more details on exact behavior. Note that you should be sure to make any value tags extend the type "value" so that they are recognized properly. Note also that only addValueArray uses this option, addValue ignores it, you should always use addValueArray if you have or could have more than one value under a particular property.

3.2.2. Runtime Variables

The following runtime variables are defined and used while the parsers execute. The following list indicated their purpose and context, and when it is safe to refer directly to them in your parser.
- **_ACTIONSET** - This is used internally by CFGXML::Parser to determine whether the action (load or save) has been set. You should not access this directly, nor should you need to worry about this variable.

- **_BOOLEANRULES** - An automatically generated hash containing one key for each boolean value, with the values pointed to being either "true" or "false" depending on whether the key represents a true or false value.

- **_BOOLEANREVERSERULES** - An automatically generated hash which is the opposite of BOOLEANRULES, where each key in the hash is a false value, and the value pointed to is the corresponding true version.

- **_CONTENTS** - If you are loading a config file, this is an array containing the lines in the config file which you can iterate through using a foreach loop. In most cases, you should read this array directly to get the contents of the file (and not open files yourself). There’s no need to modify this variable. This variable is only used during loading of config files.

- **_CURRENTSECTION** - This is an internal storage of the section the parser is currently. You should not access it directly, instead use getCurSection, enterSection, and leaveSection which handle error checking and management of the current section for you.

- **_DIRECTIVERUNCOLS** - This is used internally by makeColumnValueString to keep track of the largest value in each column when the same directive occurs multiple times in a row, such as in Common::Apache. You should not modify this array directly, it is set by makeColumnValueString as needed.

- **_DIRECTIVERUN** - If set, this value contains the name of the current property/directive which has occurred more than once in a row. You should not modify this value directly. Instead you should use resetDirectiveRun when you encounter a non-property or other situation where a "run" must end. You should not modify this value directly, it is automatically set if applicable when you call nameColumnValueString.

- **_LOADING** - A boolean flag indicating whether the parser is loading or saving. You should specify this value as an argument to run to set this value during testing (this is done for you by CFG when it runs the parser), and there is no need to access this value directly. The CFGXML::Parser module will call your load or save methods in the appropriate situations.

- **_OUTPUT** - This is a string containing the output of a parser saving a configuration file. You should normally append text to this string instead of directly writing to the files. This variable is only used during saving of config files.

- **_PROPERTYINVERTLIST** - This is an internal representation of the PROPERTYINVERTRULES option. You should not access it directly.

- **_PROPERTYRULESLIST** - This is an internal representation of the PROPERTYRULES option. You should not access it directly.

- **_PROPERTYRULESTAGS** - This is an internal list of XML tags which are property elements. You should not access it directly, instead use the getProperties, isSection, or isProperty methods of CFGXML::Parser.

- **_RECURSIVE** - Boolean whether the current instance of a parser is one which was called recursively from another parser

- **_SECTIONRULESList** - This is an internal representation of the SECTIONRULES option. You should not access it directly.
3.2.3. Public Functions

The following public functions are defined in CFGXML::Parser and can be used by parsers which extend it.

**Note:** Since the parsers are class/module-based, you must use the standard way of calling an object's own functions. This can be done in a parser by doing $self->functionName($arg1, $arg2)

3.2.3.1. Common

These functions are applicable during both loading and saving.

CFGXML::Parser new(void);
This is the constructor for CFGXML::Parser and its sub-classes. You do not normally need to call this unless you are manually testing your parser.

void run(string $action - The action the parser should perform, either load or save);
This function performs the load or save process by calling various other functions as needed, including the init function.

void init(void);
This function is called by run before loading/saving begins to set any needed options. Sub-classes of CFGXML::Parser should overload this method as described in the Options section.

boolean isSection(config4gnu::CfgObject $element);
Returns 1 (true) if $element is defined as a section tag in the current parser, 0 (false) otherwise. Note that this is not the exact opposite of isProperty. To define an element as a section, specify in the class definition file that it extends the section type.

boolean isProperty(config4gnu::CfgObject $element);
Returns 1 (true) if $element is defined as a property tag in the current parser, 0 (false) otherwise. Note that this is not the exact opposite of isSection. To define an element as a property, specify in the class definition file that it extends the property type.

boolean isComment(string $candidate - The string to be examined to see if it is a comment);
Returns true if $candidate begins with a COMMENTCHAR character possibly preceded by whitespace, false otherwise. If a quoted or escaped COMMENTCHAR appears first or immediately after initial whitespace, false will be returned.

3.2.3.2. Loading

These functions are applicable during loading.

string[] tokenize(string $input - The string to split into tokens);
Returns an array of non-empty strings containing each token. The string is tokenized by space or tab characters, and the split characters are not included in the tokens. All characters inside single or double quotes are treated as single tokens. If RIGHTCOMMENTALLOWED is true and a COMMENTCHAR is encountered at the beginning of a non-quoted token, everything including and after that character is returned as the last token, which can be tested with isComment and shifted off the array to be stored as a right comment.

void addValue(config4gnu::CfgObject $node - The node to add the value to,
string $data - The value to add to $node, boolean $multiple - Whether the property you’re adding/setting values of can have multiple values. (default = false), string $tag - The tag to use for the value tag);
Adds $data to $node using the appropriate XML tags. If $tag is specified, it is used for otherwise value is used (you should not need to specify $tag except in very strange situations). Type conversion is done automatically. Any trailing newline is trimmed, and if trimming, it is not added. If $multiple is false, any existing value will be replaced with $data, otherwise $data will be added as another value, preserving existing values.
void addValueArray(config4gnu::CfgObject $node - The node to add the values to, string[] $data - The array of values to add to $node);

Adds each item in $data as a value of $node using the appropriate XML tags. If VALUETAGS has an entry for the tagname of $node, its contents are used in order for each value entries exist in VALUETAGS than in $data, the extra tags are NOT created. If more entries exist in $data than in VALUETAGS, the last entry in VALUETAGS is used for any extra value items. This means, for example, that if a directive has a set of parameters with different meaning followed by a variable number of parameters with identical meaning, you can put in VALUETAGS an array containing one entry for each different parameter and then an extra entry to specify the tag to be used for all of the trailing identical parameters. Type conversion is done and any trailing newline is trimmed and uses are ignored, just as in addValue.

config4gnu::CfgObject newSection(config4gnu::CfgObject $parent - The element to add the new section to, string $childName - The name of the child to add (optional), string $tagName - The XML tag to use for the child (optional));

Append a new section element to $parent and returns a pointer to the new element. If no optional parameters are given, the XML tag will be section. If $tagName is given, its value will be used for the XML tag and $childName will be included in the XML but not used to determine the XML tag. If $childName is given and $tagName is not, $childName will be used to automatically determine the appropriate XML tag. Normally, you should only specify $childName and customize the XML tag using parser options.

config4gnu::CfgObject newProperty(config4gnu::CfgObject $parent - The element to add the new section to, string $childName - The name of the child to add (optional), string $tagName - The XML tag to use for the child (optional));

Append a new property element to $parent and returns a pointer to the new element. If no optional parameters are given, the XML tag will be property. If $tagName is given, its value will be used for the XML tag and $childName will be included in the XML but not used to determine the XML tag. If $childName is given and $tagName is not, $childName will be used to automatically determine the appropriate XML tag. Normally, you should only specify $childName and customize the XML tag using parser options.

config4gnu::CfgObject enterSection(string $name - The name of the section being entered);

Calls newSection to add a section named $name as a child of the current section and returns a pointer to it. This is similar to calling newSection directly except the current section and basic sanity checking is automatically maintained for you. You should use this even if your sections aren’t nested to make your parser easier to understand, just remember to leave a section before you enter the new one if they’re all at the same depth level style config files).

void leaveSection(string $name - The name of the section being left);
Checks that $name matches the name of the current section (to help detect syntax errors) and makes the current section's parent section be the new current section. If $name is not specified, no checking is performed, useful for config files which don't (i.e., INI). Note that calling leaveSection when the current section is _XMLROOT results in no change in the current section, even if called from a recursive instance of a parser whose _XMLROOT is not the true root node.

```c
config4gnu::CfgObject getCurSection(void);
Returns the current section if enterSection and leaveSection have been used in the parser.
```

```c
void endFile(string $endcomment - the ending comment text);
Appends a fileend tag to the current section, signifying the end of the file. Adds $endcomment as the comment of the fileend tag, which is automatically appended to the file during saving. All parsers should call this function when they finish, whether or not the file comment
```

```c
int locateIncludeStart(config4gnu::CfgObject $node - current section);
Returns the index to the entry after the include tag so that a subparser starts at the correct node. Must be called by a recursively called parser to find its starting point. See mon::Ini for example.
```

```c
int locateIncludeEnd(config4gnu::CfgObject $node - current section, int $curpos - current position in children list of $node);
Locates the fileend node which matches the filename of the include tag encountered at $curpos. Must be called at end of loop where a recursive parser could have been destined belonging to include file are skipped by outer parser.
```

### 3.2.3.3. Saving

These functions are applicable during saving.

```c
config4gnu::CfgObjectVector getValue(config4gnu::CfgObject $node - The node to get the values of);
Returns a list of all immediate value children of $node, doing any necessary type conversions. Note that value children are considered those which extend the type value, and is
```

```c
string getValueString(config4gnu::CfgObject $node - The node to get the value string of, boolean $delimit - whether or not to delimit the values with spaces (optional, default = 1));
```
Returns a string representation of all immediate value children of $node. Type conversion is done automatically. A space character is inserted at the beginning of the string between each value unless 0 is passed as a parameter. If no values are found, an empty string is returned.

```cpp
config4gnu::CfgObjectVector getSections(config4gnu::CfgObject $parent - The element to be searched.);
Returns a config4gnu::CfgObjectVector containing pointers to all immediate children which are fined as section tags in the current parser. If a recursive instance and was given the _XML_ROOT as its parameter, instead returns a CfgObjectVector containing only the root node, to
dling recursion easier for individual parsers.
```

```cpp
config4gnu::CfgObjectVector getProperties(config4gnu::CfgObject $parent - The element to be searched.);
Returns a config4gnu::CfgObjectVector containing pointers to all immediate children which are fined as property tags in the current parser.
```

```cpp
config4gnu::CfgObjectVector getPropertiesAndSections(config4gnu::CfgObject $parent - The element to be searched.);
Returns a config4gnu::CfgObjectVector containing pointers to all immediate children which are fined as either property or section tags in the current parser. The only tags not re
turned are data/attribute tags such as value, comment, etc. This function is iden
tical to combining getSections and getProperties except that order is preserved. It is faster:
both functions separately and combining their output, and also preserves them in proper
der, so should normally be used instead anyway.
```

```cpp
void resetDirectiveRun(void);
Resets the current directive run, if any. You should call this when a situation is en
countered where directive runs should be terminated, otherwise the run may be in
correctly assumed to continue. For example, it should be called when you enter or leave a
```

```cpp
string makeColumnValueString(config4gnu::CfgObjectVector $items, int $current);
Returns a columnized representation of the values of the element at index $cur-
current in $items. The $items variable will be automatically tested as needed to de-
tect a directive run, and the columns will be formatted to fit the largest val-
ues in each column.
```
3.2.4. Private Functions

The following private functions are defined in CFGXML::Parser and should normally be used only by
other functions defined in CFGXML::Parser. You could also use them in custom functions which you
make, or rewrites of existing functions needed for your parser. However, in both cases you should
generalize your function so that it can add to or replace existing public CFGXML::Parser functions,
unless the change is very specific to only your parser. For example, it may be necessary to extend the
_strTo and _strFrom functions to do what you need them to do.

void _setAction(string $action - Either load or save);
Sets _LOADING to 1 if $action is load, 0 otherwise. This is called automatically by run us-
ing the same parameter it was called with.

void _start(void);
If loading, reads STDIN into the _CONTENTS array and initializes the XML tree in _XML-
ROOT. If saving, reads STDIN into a XML tree. Called automatically by run.

void _finish(void);
If loading, prints _XMLROOT as a string to STDOUT unless the current instance was called re-
cursively from another parser. If saving, prints the _OUTPUT string to STDOUT. Called au-
tomatically by run.

void _fileend(config4gnu::CfgObject $node - The fileend node which signified
the end of the file);
Called automatically during saving, prints out endcomment stored in $node's com-
ment property.

string _makePaddedValueString(config4gnu::CfgObject $item - Node to get the
values of);
Returns a string containing the columnized representation of $item's value tags based on _D
RECTIVERUNCOLS, which is made automatically by makeColumnValueString, which is the only fun-
tion which should call _makePaddedValueString.

string _strFromBoolean(string $origvalue, string $booleanval);
Converts $booleanval into a string based on how $origvalue represented the value in the or-
inal config file. If $origvalue's true/falseness is the same as what is spec-
ified by $booleanval, it returns $origvalue since no conversion is necessary. Oth-
ewise, it returns the opposite version of $origvalue. You can set BOOLEANRULES to mod-
ify its behavior, you probably don't need to over-ride it.

array _strToBoolean(string $value);
Returns an array containing either "true" or "false" to indicate the true/false nature of $value for storage in the XML. You can set BOOLEANRULES to modify its behavior, you probably don't need to override it.

void _buildBooleanRules(void);
Builds the _BOOLEANRULES and _BOOLEANREVERSERULES based on the BOOLEANRULES option.

string _strFromStrin(string $value, int $quotetype);
Converts $value into a string and returns it. If $quotetype is 1 or 2, single or double quotes, respectively, are put around $value, and any internal quotes of the same type are capped.

array _strToString(string $value);
Returns an array containing the new string value and an integer representing how/if $value closing quotes are removed and escaped quotes of the same type are unescaped. For the second value, 0 = no quoting, 1 = enclosed by single quotes, 2 = enclosed by double quotes.

array _readFile(string $filename);
Returns an array containing each line of the contents of $filename. Note that the CFG library may rewrite the filename.

void _writeFile(string $contents, string $filename);
Writes $contents to $filename. Note that the CFG library may rewrite the filename.

3.3. config4gnu::CfgObject

The config4gnu::CfgObject module provides a wrapper around the actual XML library being used by the parsers. You can use its functions to read and write attributes of properties and sections in your parsers.

Note: While you may often use config4gnu::CfgObject's functions in your parsers to do simple things, for anything other than basic read/write of single values, first check CFGXML::Parser as it contains many functions which do fancy things for you, and does them in a consistent way.
3.3.1. Loading

The following functions are useful during loading of files and creation of the XML tree.

```c
config4gnu::CfgObject addContainer(config4gnu::CfgObject $parent - The parent
container to add the new child to, string $childTag - The XML tag to use for
the child being added, string $childName - The name of the child being added
(optional));
```

Creates a new container element (property or section), appends it to $parent, sets its name
Name if provided, and returns a pointer to it. This function should normally only be collec
tions.

```c
void addComment(string $data - The comment to add to the current
config4gnu::CfgObject);
```

Adds $data to the current config4gnu::CfgObject using the appropriate XML tags. Any trail-
ing newline is trimmed, and $data is added regardless of whether it is empty.

```c
void addEndComment(string $data - The end comment to add to the current
config4gnu::CfgObject);
```

Adds $data to the current config4gnu::CfgObject using the appropriate XML tags. Any trail-
ing newline is trimmed, and if $data is empty after trimming, it is not added. Note that Par-
ing comments of a file and terminate it.

```c
void addCloseComment(string $data - The close comment to add to the current
config4gnu::CfgObject);
```

Adds $data to the current config4gnu::CfgObject using the appropriate XML tags. Any trail-
ing newline is trimmed, and if $data is empty after trimming, it is not added.

```c
void addRightComment(string $data - The right comment to add to the current
config4gnu::CfgObject);
```

Adds $data to the current config4gnu::CfgObject using the appropriate XML tags. Any trail-
ing newline is trimmed, and if $data is empty after trimming, it is not added.

```c
void addWhitespace(string $data - The whitespace to add to the current
config4gnu::CfgObject);
```

Adds $data to the current config4gnu::CfgObject using the appropriate XML tags. Any trail-
ing newline is trimmed, and if $data is empty after trimming, it is not added.

```c
void _addDataTag(string $data - The tag to use for the added data
config4gnu::CfgObject, string $tag - The tag to use for the added data
```
element, string $attr - The type attribute of the tag to be created to hold the data (optional));
Adds $data to the current config4gnu::CfgObject using $tag for the XML tag. If $attr is given, tribute of the XML tag is set to its value. Any trailing newline from $data is trimmed, after trimming, it is not added unless $tag is comment. You should not normally call this function directly.

3.3.2. Saving

The following functions are useful during saving of files and reading of the XML tree.

config4gnu::CfgObjectVector getChildren(void);
Returns all children (properties, sections, and data tags) of the current element.

config4gnu::CfgObjectVector getElementsByTagList(string[] $tagList - The list of tags to try to match);
Returns all immediate children whose tags are found in $tagList in the order they are found.

config4gnu::CfgObjectVector getComments(void);
Returns a list of all immediate comment children of the current element.

string getCommentString(void);
Returns a string representation of all immediate comment children of the current element. A newline character is inserted at the end of each comment found. If no comments are found, an empty string is returned.

config4gnu::CfgObjectVector getRightComments(void);
Returns a list of all immediate right comment children of the current element.

string getRightCommentString(void);
Returns a string representation of all immediate right comment children of the current element. Nothing is inserted between, before, or after each right comment. If no rig ments are found, an empty string is returned.

config4gnu::CfgObjectVector getEndComments(void);
Returns a list of all immediate end comment children of the current element.
string getEndCommentString(void);
Returns a string representation of all immediate end comment children of the current element. Nothing is inserted between, before, or after each right comment. If no end remarks are found, an empty string is returned.

config4gnu::CfgObjectVector getCloseComments(void);
Returns a list of all immediate close comment children of the current element.

string getCloseCommentString(void);
Returns a string representation of all immediate close comment children of the current element. Nothing is inserted between, before, or after each right comment. If no close remarks are found, an empty string is returned.

config4gnu::CfgObjectVector getWhitespaces(void);
Returns a list of all immediate whitespace children of the current element.

string getWhitespaceString(void);
Returns a string representation of all immediate whitespace children of the current element. Nothing is inserted between, before, or after each right comment. If no whitespace are found, an empty string is returned.

string getName(void);
Returns the name of the current element if it has one, otherwise its XML tag name is returned.

string get_property_tagname(void);
Returns the XML tag of the current element.

string getData(void);
Returns the string data contained in the current element, or an empty string if no data is
3.4. config4gnu::CfgObjectVector

config4gnu::CfgObjectVector is a wrapper around the underlying XML library which you can use to access data from the config4gnu::CfgObjectVectors returned by some functions.

config4gnu::CfgObject get(int $index - The index of the element to return, between 0 and length-1); Returns the element at $index.

void put(void); Adds the element to the end of the config4gnu::CfgObjectVector.

int size(void); Returns the number of elements in the current config4gnu::CfgObjectVector. Keep in mind that dexed starting at 0.

3.5. An Example Parser

This is an example of a parser which uses many of the functions available in the Perl modules to load and save apache-style configuration files (such as those used by Apache and ProFTP). It would then be extended in specific parsers for Apache, ProFTP, and others by setting options in the init functions of parsers which extend it. These settings should be sufficient to handle all aspects of both files. If they are not, then this parser may be modified to be more flexible (to avoid copying and pasting of the entire parser to modify a few lines).

Example 2. The Common::Apache Parser

package CFGXML::Parser::Common::Apache;
use strict;
use CFGXML::Parser; Make CFGXML::Parser available to this module

@CFGXML::Parser::Common::Apache::ISA = qw(CFGXML::Parser);
Define CFGXML::Parser as the parent module of this module

#NOTE: This parser is for apache *style* files (syntax), NOT for
# configuration files for apache. It should be subclassed to add
# support for Apache, Proftfp, and friends.

sub init Set CFGXML::Parser's options in this function
{
  my $self = shift;
  $self->SUPER::init(); Call init function of CFGXML::Parser
  $self->[ROOTTAG] = 'apache-style'; Set the tag of the root XML element to "apache-style"
  $self->[RIGHTCOMMENTALLOWED] = 1; By default, right comments are allowed in apache-style
$self->[CLOSINGCOMMENTALLOWED] = 1; By default, close comments are allowed as well

sub load
  This function converts the config file to XML
  {
    my $self = shift;
    my $value;
    my $rightComment;
    my $comment;
    my $prop;
    my $curSec;
    my $whitespace;
    my @toks;

    foreach (@{$self->/_CONTENTS/}) Iterate through each line of the config file
    {
      if (/^\s*(\^\s/(.*))>(\s*\#.*)?/) This matches the beginning of a section, such as <VirtualHost>
      {
        section start encountered
        $rightComment = $2;
        $whitespace = $1;
        @toks = $self->tokenize($2); Splits the value(s) found at non-quoted whitespace
        my $secName = shift @toks; The first value is always the name of the section

        $curSec = $self->enterSection($secName); Create a new section named $secName and make it the current section

        $self->addValueArray($curSec, @toks); Add a new value tag for each item in the @toks array

        add rightcomment & comment if allowed
        if ($self->[RIGHTCOMMENTALLOWED])
        {
          $curSec->addRightComment($rightComment); If $rightComment is empty, this function will ignore it
        } else
        {
          die "Right comments are not allowed in this file format";
        }
        $curSec->addComment($comment); Again, no need to check that $comment isn’t empty, this is done for you
        $comment = "";

        $curSec->addWhitespace($whitespace);
      } elsif (/^\s*<\/(.*>)>(\s*\#.*)?/) {
        section end encountered, append any endcomment & leave it
        my $closecom = $2;
        $self->getCurSection->addCloseComment($closecom);
        $self->getCurSection->addEndComment($comment);
        $comment = "";
        $self->leaveSection($1); Check if $1 is the name of the current section & make its parent the current section
    }
  }
}
} elsif (/^\s*#/ | /^\s*$/) 
  {$comment = $_;}
elsif (/^\s*\{\s*/)
  {$
directive encountered
$whitespace = $1;
@toks = $self->tokenize($2);
tokenize the part of the line after the whitespace
my $propName = shift @toks;
my $curProp = $self->newProperty($self->getCurSection, $propName); Create a new property named $propName as a child of the current section

add value tags for each parameter of the directive
if ($#toks >= 0)
  {
#last token might be the right comment
if ($self->isComment($toks[$#toks])) The last token may be a right comment
  {
    my $rc = pop @toks;
    $curProp->addRightComment($rc);
  }
$self->addValueArray($curProp, @toks);
}

$curProp->addComment($comment);
$comment = "";
$curProp->addWhitespace($whitespace);
}
else This matches any other config line that isn't formatted properly
  {
    die "$_ . 'is not a recognized config line';
  }
$self->get(_XMLROOT_)--addEndComment($comment); Add any remaining comments to the root XML element
}

sub save This function converts the XML back into the config file
  {
    my $self = shift;
    
    #unparse XML back into config file

    $self->saveSectionsAndProperties($self->_XMLROOT_);
    $self->{_OUTPUT} .= $self->{_XMLROOT_}->getEndCommentString;
  }

sub saveSectionsAndProperties A recursive function which converts each section and its children
  {
    my $self = shift;


my $node = shift;
my $sectprops = $self->getSectionsAndProperties($node);
$self->resetDirectiveRun(); # Directive runs should not continue into a new section
for (my $i = 0; $i < $sectprops->size(); $i++)
{
    my $item = $sectprops->get($i); # Retrieve item $i from the config4gnu::CfgObjectVector
    if ($self->isSection($item))
    {
        $self->{_OUTPUT} .= $item->getCommentString;
        $self->{_OUTPUT} .= $item->getWhitespaceString;
        $self->{_OUTPUT} .= ' <'. $item->getName;
        $self->{_OUTPUT} .= $item->getValueString . ' >';
        $self->{_OUTPUT} .= $item->getRightCommentString . "\n";
        $self->saveSectionsAndProperties($item); # Recursively call this function to display children who are sections
    }
    else
    {
        $self->{_OUTPUT} .= $item->getCommentString;
        $self->{_OUTPUT} .= $item->getWhitespaceString;
        $self->{_OUTPUT} .= ' </'. $item->getName . ' >';
        $self->{_OUTPUT} .= $item->getCloseCommentString . "\n"; # Display closing tag and end/close comments of the section
    }
}
else
{
    $self->{_OUTPUT} .= $item->getCommentString;
    $self->{_OUTPUT} .= $item->getWhitespaceString;
    $self->{_OUTPUT} .= $item->getName;
    $self->{_OUTPUT} .= $self->makeColumnValueString($sectprops, $i); # Automatically detect directive runs and format values into columns as needed
    $self->{_OUTPUT} .= $item->getRightCommentString . "\n";
}
}

1; Perl modules must return 1 when run

3.5.1. An example subclass

This is an example of a parser which subclasses Common::Apache to parser files specific to a fictitious application, BobServer. BobServer is a webserver which displays directly listings using Color1 for text and Color2 for background. BobServer's configuration includes one main section, BobServer, and any number of possibly nested directory Dir sections. Color settings are inherited from the parent directory unless explicitly set to something different. Allowed directives are HostName, PortNumber, Color1 and Color2, and the only allowed section is BobDir. Close comments are not allowed, but end comments are.
Example 3. The BobServer Parser

```perl
package CFGXML::Parser::App::Fictitious::BobServer;
use strict;
use CFGXML::Parser::Common::Apache; Make CFGXML::Parser::Common::Apache available to this module

@CFGXML::Parser::App::Fictitious::BobServer::ISA =
  qw(CFGXML::Parser::Common::Apache);
Define CFGXML::Parser::Common::Apache as the parent module of this module

sub init { Set options specific to BobServer in this function
    my $self = shift;
    $self->SUPER::init(); Call init function of parent class
    $self->{ROOTTAG} = 'bobserver-config'; Set the tag of the root XML element to something unique to this parser
    $self->{CLOSECOMMENTALLOWED} = 0; By default, close comments not allowed as they normally are in Common::Apache
    $self->{PROPERTYRULES} = {
        Color1 => ['Colro1'],
        Color2 => ['Colro2']
    }; Since Bob often mistypes the word Color, he has made the Colro1 directive synonymous with Color1, and Colro2 synonymous with Color2. It is important that synonyms be listed in PROPERTYRULES so that they are treated as the same setting in other layers of CFG

    }
    1; Perl modules must return 1 when run

With these few minor option changes, the generic Common::Apache parser for apache-style config files can be used for BobServer's configuration.

4. Adding Entities

This section is currently a work-in-progress. If you are interested in adding support for more entities (other applications, etc.) please email us and we will help you out.

4.1. About the Class Definition File

The class definition file is a Config4GNU-specific file that describes the various elements that occur in the XML representation of a particular type of configuration file. The class definition files are found in config4gnu/data/classes. They contain one or more type definitions. Each type definition defines attributes and child elements that may appear in an element of that type. Type definitions may also identify themselves as members of a particular class of objects. Finally, type definitions may define the type of object content that objects may contain.
4.2. Layout of a Class Definition File

The root element should be `type-definitions`. In side the `type-definitions` element there should be one or more `type` elements. Each `type` element corresponds to one type definition.

Here is a short example:

**Example 4. A type definition file**

```xml
<?xml version="1.0"?>
<type-definitions>
  <type name="date">
    <properties>
      <property name="month" minOccurs="1" type="builtin:integer">
        <name>Month</name>
      </property>
      <property name="day" minOccurs="1" type="builtin:integer">
        <name>Day</name>
      </property>
      <property name="year" minOccurs="1" type="builtin:integer">
        <name>Year</name>
      </property>
    </properties>
  </type>
</type-definitions>
```

4.3. Defining a type

The `name` attribute of the `type` element is the name of the element. This is how the type will be represented and identified internally.

The `properties` element contains zero or more `property definitions`. Property definitions define what child elements (and attributes) may occur in an element of this type. Inside the `properties` element include a `property` element for each property definition. See also `Property Definitions`.

The `extend` element identifies a `IS A` relationship. This has the effect of importing all the property definitions from the specified type into the current type. It also allows classifying a type as a member of other classes. See `Classifying a type`.

The `form` element associates elements of a particular type to form definition files that are appropriate for configuring that type. Use the `form` element's `filename` attribute to specify a filename. E.g.
Example 5. Specifying the form used to display a type

<form filename="sambashare.xform">

4.4. Classifying a type

There are a few predefined types available for classifying types. These types include property and section. These types do nothing in themselves, but are types that frontends will look for when trying to decide how to display certain information. The following paragraph describes how the Config4GNU GUI client uses the predefined type classes.

The property type identifies an object that should appear in the property list for an object when it is selected. The property list is found under the Properties tab.

The section type identifies an object that should appear in the tree view in the left pane in the GUI client. It also identifies objects that should appear as children of the current object.

To use the type classes, include the extend element in the type definition. E.g.

Example 6. Type that extends both property and section

<type name="SamplePropertyAndSection">
  <extend type="builtin:property"/>
  <extend type="builtin:section"/>
</type>

4.5. Property Definitions

A property element identifies an element or attribute that can exist inside an object. All property definitions have a number of properties:

- owner type - is the type of object this property can exist in
- tagname - is the identification of the property. The tagname is the name of the element when the property is represented as an element or the name of the attribute with a @-sign prefix when represented as an attribute.
- name - this is the human-readable name of the property. Names can be supplied for every language supported.
- description - is a human-readable description of the property.
• type - is the type of the property (not to be confused with the type of object that contains this property)
• minOccurs - defines the minimum number of times this property must occur for the containing object to be valid, e.g. 0 or 1.
• maxOccurs - defines the maximum number of times this property is allowed to occur. Use unbounded if there is no limit.

Do not explicitly specify the owner type in the property definition; this is implied by which type definition it occurs in.

The tagname is specified using the name attribute of the property element.

Other properties identified using attributes of the property element are minOccurs, maxOccurs, and type.

Any properties that can be translated to multiple languages must be defined using elements. This way, we can use the xml:lang attribute to identify the target language.

Glossary of Terms

**configuration entity**

A class of node containing a set of configuration properties for a particular thing. For example, the Apache webserver's configuration is a configuration entity.

**object content**

An object's content, or *value*, is the value of the XML attribute if the object is represented by an attribute, the content of an XML element if the object is represented by a simple element, or the value of the child element named *value*.

**type definition**

An XML file in config4gnu/data/classes/ which defines how a top-level XML element in a configuration entity should be handled by CFG.
1. Introduction

1.1. Purpose

The purpose of this specification is to precisely document the syntax and semantics of the XML format that is used to communicate between the Config4GNU subsystem and the underlying configuration file parsers.

1.2. Definitions

Config4GNU subsystem

Also known as the "middle layer," this component of the Config4GNU system exists in between the front-ends and the back-ends. It provides routines that are common to all front-ends so that code does not need to be duplicated. It automatically invokes the necessary back-ends when configuration files need to be read or written.

Configuration file parsers

Also known as the "back-ends," these programs read and/or write native-format configuration files. They translate between a standard XML format, which is described in this document, and the native format of the configuration file. They are automatically invoked by the middle layer as needed.

Nodes

Identifies some aspect of the configuration. It is a location on the configuration hierarchy that Config4GNU presents to the user. As part of a hierarchy, it has one "parent" node and zero or more "child" nodes. Is also composed of properties.
Figure 1. Node Hierarchy

Root Node "localhost"

Daemons

Apache  Samba

System Settings

Property

This is one configuration setting. Properties are attached to a configuration node. Properties are composed of a name and a value.

2. An example

To help understand the XML structure specified by this document, please examine the following INI-type file and the resulting XML specification.
Example 1. An INI-style configuration file

[global]
local master = false
os level = 0
wins server = 153.42.16.10
security = share
workgroup = mellinger

[public]
path = /home/jason/public
writeable = false
guest ok = true

Following is the equivalent XML representation of this data.

Example 2. The XML representation of the INI file

<inifile>
  <inisection sectionname="global">
    <property name="local master"><value>false</value></property>
    <property name="os level"><value>0</value></property>
    <property name="wins server"><value>153.42.16.10</value></property>
    <property name="security"><value>share</value></property>
    <property name="workgroup"><value>mellinger</value></property>
  </inisection>
  <inisection sectionname="public">
    <property name="path"><value>/home/jason/public</value></property>
    <property name="writeable"><value>false</value></property>
    <property name="guest ok"><value>true</value></property>
  </inisection>
</inifile>

This XML representation contains one root node with two child nodes. The two child nodes represent the two sections in the INI file. Attached to the two sections are various properties. The following diagram shows this data model.
3. Document Structure

3.1. Root element

The root element identifies the node type. The name of the root element must match the type of the configuration node that the document represents. For instance, when the INI file parser generates an XML representation of an INI file, it creates a root element named inifile, because that is the type of node that represents an INI file.

The root element will contain child nodes and properties. See the following sections for an explanation of child nodes and properties.
3.2. Child nodes

Child nodes are identified by looking for child elements of the element for the current node with an attribute sectionname. The name of the child element identifies the node type of the child node. The value of the sectionname attribute identifies the name of the node. For example, `<insection sectionname="global">` identifies a child node of type insection with the name "global."

Child nodes themselves can contain additional child nodes and properties.

3.3. Properties

Properties are identified by looking for child elements with the name of property or parameter.

The property element can contain one or more of the following entities:
- value
- comment
- whitespace

These elements are simple elements. They can only contain text. They are described in further detail in a later section.

4. Properties

This section describes elements and attributes used to identify properties, and the various components of properties.

4.1. An example property

The following XML excerpt is a setting that could be found in an INI file. It shows use of the property element and all three elements that a property can contain.

Example 3. A property

```xml
<property name="path">
  <whitespace>
  </whitespace>
  <value>/home/jason</value>
  <comment type="append">My home directory</comment>
</property>
```
4.2. The property element

The property element identifies a Config4GNU property. The name of the property is identified using the name attribute. In the example above, the name of the property is "path."

4.3. The value element

The value element identifies the property’s value. Since it can be anything, it is implemented as a simple text element rather than an attribute. If the value can contain linefeeds, the parser may implement the contents of the value element as an XML CDATA section.

4.4. The comment element

The comment element identifies a user added comment on the property. The comment is there in case the Config4GNU front-end chooses to display the information to the user, but it mostly exists so that when the configuration file is rewritten, the comment will be preserved.

Parsers may specify attributes on the comment element to give itself additional information on how the comment should be rewritten to the file. In the example above, the comment element has an attribute "type" set to "append." This information tells the configuration file writer that the comment should be written on the same line as the property, "appended" onto the end of the line.

Multiple comments may appear in a property element. The comment element is not required to appear. If no comment element is specified, the file writer is to assume no comment was present in the file.

4.5. The whitespace element

The whitespace element identifies any ignorable whitespace that existed in the file before this element. The whitespace information is not used by the front-end at all; it exists only so that the rewritten configuration will not contain artificial changes. If special characters occur in the whitespace, it may be necessary to create a CDATA section to contain the contents of the whitespace.

In the example above, the whitespace element tells the file writer to include eight spaces prior to the property statement in the configuration file.

Like the comment element, attributes can be used to provide additional information. Also like the comment element, more than one whitespace element can exist for a property.
If no whitespace element is present, or a whitespace element with a particular attribute is not present, the configuration file writer is to assume a "default" value for the whitespace. For example, if no whitespace element is present, the writer may choose to indent the property by eight spaces because all other properties in the section were indented by this amount.

5. Final thoughts

5.1. Weaknesses

Currently, this specification does not provide a way for comments or whitespace to exist separate from a property.

Index