A configuration system for Siemens Airlink

Søren Løvborg

Abstract

Siemens Airlink is a railway technology providing secure wireless connectivity between rolling stock and wayside equipment. This report documents the process of designing, implementing, and testing NMS-NG, a replacement for the aging Airlink NMS configuration system.

The result is a highly flexible configuration system and matching security model, specifically tailored for the daily workflows of the different Airlink user segments. While additional work is required (particularly on integration), the NMS-NG documented in this report represents an important step towards providing an effective platform for coming generations of the Airlink system.

Kongens Lyngby 2012
IMM-M.Sc-2012-100

Technical University of Denmark
Department of Informatics and Mathematical Modelling
Building 305, 2800 Kongens Lyngby, Denmark
Phone +45 4525 3351, Fax +45 4588 2673
reception@imm.dtu.dk
www.imm.dtu.dk

Contents

1 Introduction

Modern railway systems require the rolling stock to be in constant radio contact with wayside equipment, for purposes such as:

- Train protection and control
- TV surveillance
- Passenger information
- Public access Internet service for passengers

A range of technologies are used to provide the above, ranging from simple balise and inductive loop technologies to the more recent GSM-R standard.

Siemens Airlink is such a technology, providing a secure wireless connection between the on-board and wayside computer networks. Airlink only provides connectivity up to the network transport protocol layer, unlike e.g. GSM-R, which also defines application layer protocols (e.g. voice calls).

Airlink is used in numerous installations worldwide for transmission of train control signals and passenger information, and is an especially popular choice in new metropolitan rail installations (figure 1).

In Denmark, Airlink technology has previously been used to provide high-speed wireless Internet service to S-train passengers. The on-going S-bane signalling upgrade program will see its scope expanded to include train control signals and passenger information between 2014 and 2018.

The Airlink system uses off-the-shelf 802.11n wireless technology for communication between train units and access points. The primary advantage of this approach compared to the current GSM-R standard is higher bandwidth, something which e.g. is a real concern with GSM-R in the Copenhagen metropolitan area due to the density of the train traffic. [Kroyer08]

Location	Commissioned	Daily passengers
Copenhagen, Denmark (S-trains)	2011	300,000
Paris, France	2011	725,000
São Paulo, Brazil	2010	~ 900,000
Algiers, Algeria	2010	~ 300,000
Budapest, Hungary	2008	500,000
Guangzhou, China	2006, 2010	~ 1,000,000

Figure 1: *A sample of locations where an Airlink system has been either comissioned or already implemented. (Source:* [Siemens12] *and operator websites.)*

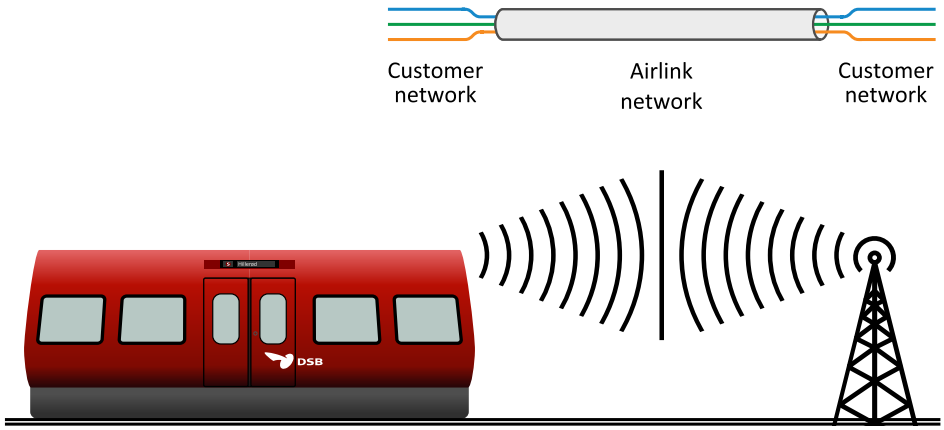


Figure 2: *A sketch of the logical (top) and physical (bottom) Airlink system.*

1.1 Description of the Airlink system

From the customer's point of view, Airlink simply provides a number of parallel network bridges, each with one endpoint on the rolling stock and another in a central server room.

Under the hood, the system is of course a lot more complex.

At one end of the system, on the rolling stock, we have various separate customer networks (e.g. one network for train protection, at least one for other service functions, and one for public access).

These on-board networks are connected to an Airlink *train unit* (TU), which communicates with stationary *access points* (APs) along the track.

At the other end of the Airlink system, the signals are received by a *central system router* (CSR) in a central server room, which connects to a set of physical customer networks matching those on the rolling stock.

Typically, the customer networks are kept physically separate at each end for security and reliability, e.g. preventing a faulty network card in a passenger's laptop from interfering with the train protection system and bringing the train to a halt.

Though the Airlink system multiplexes these networks physically out of necessity, one of the most important duties of the Airlink system is to maintain the network integrity as if the networks were physically separate all the way. This means not only keeping the networks logically separate (as in VLANs), but also enforcing strict quality-of-service requirements to ensure e.g. that train protection signals always get sufficient bandwidth.

1.2 About the Airlink NMS

Pivotal to the Airlink system is the NMS (Network Management System), responsible for configuration and monitoring of the Airlink network.

For the *next generation NMS* (NMS-NG), Siemens desires a complete rewrite, taking into account the lessons of the current NMS.

The present NMS is a separate network node, hosting a web application allowing users to monitor SNMP variables and traps from the other nodes in the network, as well as configure the Airlink system.

Originally developed by a third-party contractor, the current NMS implementation uses a diverse and complex mix of languages and technologies in what can best be described as spaghetti code. For these reasons, it has not been properly updated in a while and is beginning to show its age.

The biggest changes in scope and purpose (requested by the Airlink development group) are:

- NMS-NG should exclude the monitoring aspect, since all deployments have in practice used industry standard monitoring software such as Nagios, rather than the rudimentary monitoring capabilities of the NMS.
- NMS-NG should not be a designated "box", but a piece of software that can be used on multiple computers.

For these reasons, it is useful to think of NMS-NG simply as "the Airlink configuration system".

Configuration deployment

To deploy a new configuration, the NMS does not communicate directly with nodes in the Airlink system. Instead, the NMS sends the new configuration to the CSR, which is then responsible for applying it across the Airlink system.

2 Requirements analysis

2.1 User segments

Essential to analysing the project requirements are the three Airlink user segments:

The Airlink development group in Denmark develops the core Airlink technology. They do not interact directly with customers. NMS-NG is used for development and testing.

The development group needs maximum flexibility in configuring the Airlink system, including the ability to tweak system internals.

The sales and support group in France uses NMS-NG to customize and configure the Airlink technology for the specific requirements of each customer.

Sales and support works with Airlink at a higher level, using an interface provided by Airlink development.

The customer may need to use NMS-NG occasionally in their daily operations, e.g. when replacing a defective train unit or access point.

In some cases, such tasks are handled by the local Siemens office, in which case the local office will be considered the customer for the purposes of this project.

2.2 Modularity requirements

The Airlink configuration system must enable a high level of configuration re-use.

Configuration re-use within a deployment
The network nodes (train units and access points in particular) in an Airlink deployment can easily number in the thousands. Each node is individually configurable, with more than two hundred configuration options per node. The configuration system must provide a way to organize these nodes into configuration classes sharing all or parts of their configuration as necessary.

Configuration re-use across deployments
Airlink is deployed in dozens of locations worldwide, and each deployment must be supported for decades. The configuration system must enable changes made by Airlink development to be easily passed first to sales and support, and then to every customer, without conflicting with existing customizations and without unintentional side effects.

2.3 Security requirements

For analysing the security requirements, consider the three classical CIA security objectives and the four different groups of threat agents seen in figure 3.

Threat agents	Security objectives		
	Confidentiality	Integrity	Availability
Internal (Siemens employees)			
Customers			
Third-parties			
Acts of nature			

Figure 3: *Security objectives and threat agents for NMS-NG.*

It is important to emphasize that a threat agent is not necessarily malicious, and protection against accidental threats is as important (if not more important) as protection against malicious threats.

Confidentiality and availability are larger issues of the Airlink system, and outside the scope of this project. Similarly, a number of integrity issues (e.g. physical security) are also out of scope.

NMS-NG must, however, secure the integrity of the Airlink configuration system, i.e. ensuring that all changes to the configuration is subject to a defined security policy, which among other things determines which configuration properties a given user can change, as well as the allowed values.

2.4 Interface requirements

Siemens interface The development group and the sales and support group share interface requirements. The Siemens interface must:

- allow full access to (and control of) the configuration system,
- support the use of version control for both generic and deployment-specific configurations,
- be scriptable to enable reproducible test environments and automatized regression tests.

These requirements call for a text-based configuration format that can be used with version control systems already in use by Siemens, as well as a command-line tool for dealing with configuration management.

Customer interface As mentioned above, the customer is quite limited in their interactions with the system. The customer interface must:

- allow basic manipulation of nodes (e.g. configuring the hardware model or location of a node),
- be user friendly and *not* require the use of a command-line,
- permit remote access (subject to security policy); the customer should not need to be physically present in the server room when upgrading equipment in the field.

These requirements call for a simple web interface with a limited configuration file editor. A web interface is easily remotable, and can be secured using separate off-the-shelf software.

3 Design

Remark The following sections contain a number of entity-relationship diagrams, as described in [Chen76]. A quick recap for readers unfamiliar with the diagram type: Designed for modelling relational databases, E-R diagrams provide an abstract description of data models. Figure 4 shows the primary building blocks: Entity sets (corresponding to database tables, record types or classes), relationship sets (describing relational semantics between entity sets), and attributes (corresponding to database columns, record fields or class properties), which can be associated with either entities or relationships.

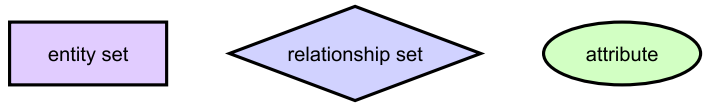


Figure 4: *An overview of symbols in an entity-relationship diagram.*

3.1 Configuration objects

The current NMS defines several different types of configurable entities:

- Nodes (or *units*)
- Locations
- Layers
- Configuration groups
- Initialization scripts
- Cryptographic tokens
- Firmware images
- Properties (or *variables*)

The diagram in figure 5 illustrates how the NMS data model has a number of predefined attributes, while the existence of a generic property system provide extensibility.

The NMS-NG datamodel (figure 6) consolidates this into a simpler and more flexible configuration model, in which *all* configuration data is stored using a generic property system. To manage the increased number of generic properties, a OOP inspired class-based inheritance system is introduced as a replacement for the various grouping features of the existing system. This ensures greater flexibility and reduces the overall complexity of the configuration system.

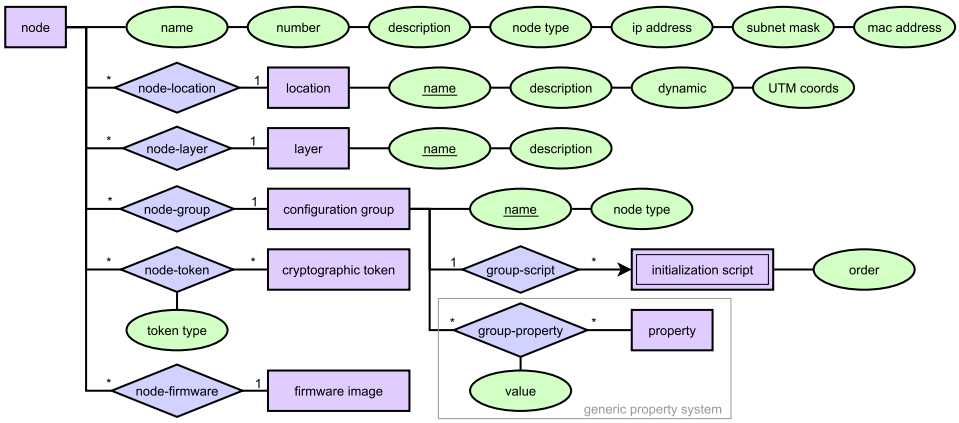


Figure 5: *The entity sets of the current Airlink system.*

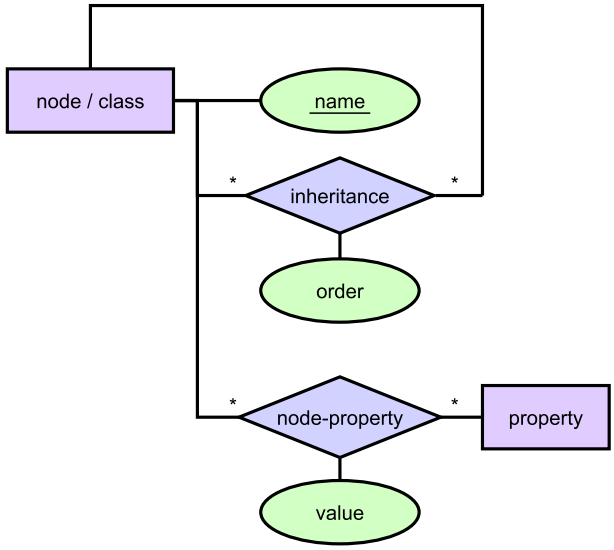


Figure 6: *The entity sets of the NMS-NG datamodel.*

Here follows a brief overview of the different entities, and their role in the system.

Nodes

Nodes refer to the individual computers making up the Airlink system, such as access points, train units and routers. The goal of the configuration system is to assign a customized configuration to each individual node.

The current Airlink system refers to these entities as *units*. Since the term is also used by the configuration system in the *unit of measurement* sense, NMS-NG uses the unambiguous (and more recognizable) term *node* instead (as in *network node*), though the term *unit* remains in a few places for compatibility with the existing system.

Locations, layers and configuration groups

The current NMS assigns each node exactly one *location*, *layer* and *configuration group*, each of which provide settings common to multiple nodes.

NMS-NG replaces this rather inflexible system with the generic class system, allowing arbitrary groupings.

Initialization scripts and properties

Each configuration group is assigned a set of *initialization scripts*, which are concatenated (in a defined order) and executed on each member node during boot. NMS-NG automatically concatenates the configuration scripts into a single value, and treats the result like any other generic property value.

Besides executing various shell commands, the initialization scripts of the current Airlink system are also responsible for setting up *variables* on the node.

Since the variables are set up using shell commands, the only way to determine the variables configured for a given node is by executing the shell scripts on that node. This gives great flexibility, but also makes it harder to reason about node configurations. For this reason, NMS-NG moves variable assignment out of the initialization scripts, makes them first-class citizens of the configuration, and determines all values *before* deployment.

In accordance with the new OOP inspired model, to distinguish them from ordinary environment variables, and because these values are actually constant for a given configuration and node, NMS-NG will use the term *property* instead of *variable*.

Cryptographic tokens

Each node is assigned a number of cryptographic tokens. The tokens are opaque binary data, stored in individual files as hexadecimal strings.

NMS-NG allows these tokens to remain as individual files or to be stored as tabular data in a single CSV file.

Firmware images

Each node is assigned a specific firmware image. The images consist of opaque binary data, stored in individual files.

NMS-NG treats the firmware as a generic property value, but the handling of firmware is otherwise essentially unchanged.

3.2 Inheritance

As seen in figure 5, the existing NMS groups nodes by *location* (physical location), *layer* (logical location), and *configuration group* (themselves subdivisions of a specific node type). Each grouping provides values for a fixed set of attributes. Since each group is entirely independent of the others, configuration values can only be shared between groups by explicit (and error-prone) repetition.

In practice, an Airlink deployment may call for other groupings, not supported by this limited approach:

- Train units on trains servicing different train lines may require different configuration options.
- To limit interference and maximize bandwidth, it might be desirable to change radio channels between alternate access points along a track.
- A node may be replaced by a newer hardware model, requiring different configuration options (firmware in particular) for that particular node.
- Troubleshooting may call for a limited patch roll-out to a chosen subset of nodes.

Each of these groupings are orthogonal to one another, and a bad fit for the available groupings of the existing NMS.

In the current Airlink system, the work-around has been cut-and-paste: In one small example configuration provided by the Airlink development group, 26 % redundancy was observed across five configuration groups. Two of the groups differed only in their choice of radio channels, with the remaining 98 % of the group configuration needlessly repeated due to insufficient grouping mechanisms.

NMS-NG solves these problems by introducing a flexible class system (figure 7), enabling groups to be split into subgroups and allowing arbitrary attribute assignments for each group. Support for *multiple inheritance* enables orthogonal groupings (figure 8).

In object-oriented programming, multiple inheritance has gotten a bad rap for two reasons:

Multiple inheritance is more complex than single inheritance

As just demonstrated, for NMS-NG the complexity is largely necessitated by the need for orthogonal groupings, a feature that is simply not supported by single inheritance. An alternative could be orthogonal groupings without inheritance, but there is a reason why OOP reigns as the most widespread programming paradigm: Inheritance provides a more elegant and natural fit for the way most people think about data.

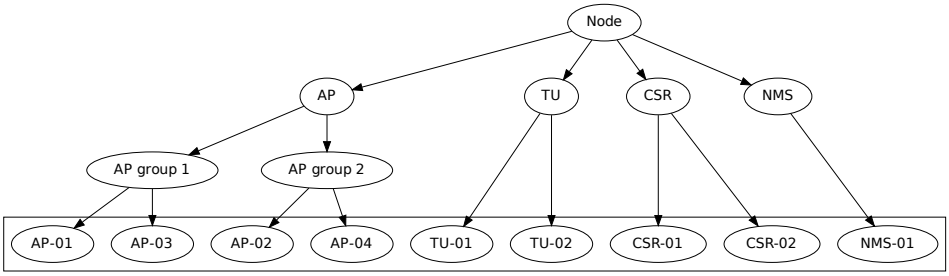


Figure 7: *A class graph resembling the fixed groupings of the existing NMS.*

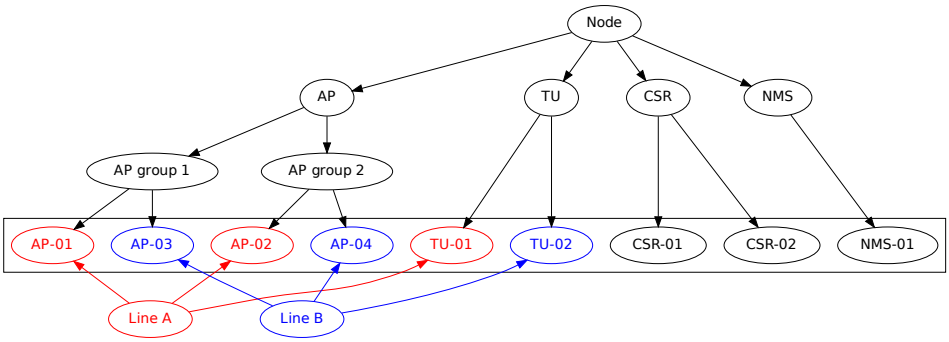


Figure 8: *A multiple inheritance class graph, demonstrating orthogonal groupings by* line *and* AP group.

While multiple inheritance certainly allows users to construct incredibly complex inheritance graphs, it doesn't force them to. In the end, multiple inheritance is a scalable technology, giving the users the exact amount of complexity they ask for.

The complexities of *implementation* will be discussed in section 5.

Multiple inheritance has worse performance than single inheritance

In the case of native compilation (e.g. C++), multiple inheritance often introduces a slight performance penalty at runtime, compared to simple vtable-based implementations of single inheritance. This does not apply to NMS-NG, as everything is resolved at compile-time.

3.3 Properties

The available properties are defined in individual property files, which specify data types and other contraints, along with descriptive text for the property. Besides validation, the constraints may of course also be used by a GUI frontend to assist the user in entering property values.

Referencing a property name with no associated property file will cause a warning.

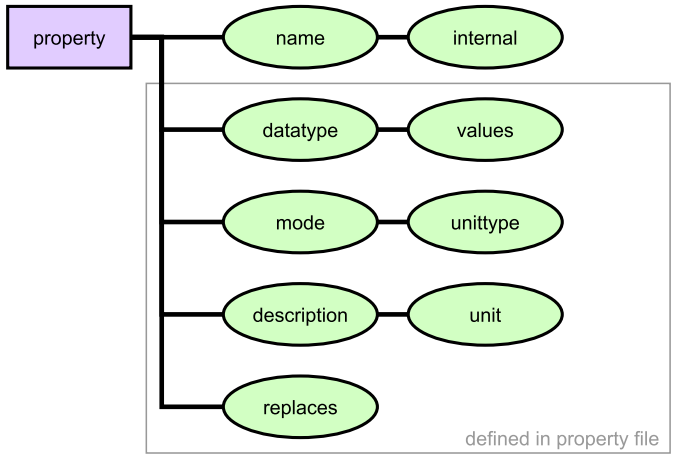


Figure 9: *The NMS-NG property entity.*

Wildcard property names

When defining properties, one or more dotted elements of the property name may be an underscore (`'_'`). This is a wildcard which may stand for any symbol (a string matching the regular expression `/[_a-zA-Z0-9]+/`).

The interpretation is that the property definition applies to all property names that match the given pattern.

An concrete example is the `multiplexer.feature.datastream.outbound._` property, which may be defined for priorities `p0` through `p15`. (The compiler does not verify that properties restrict themselves to these 16 names, though, as the wildcard can stand for any symbol.)

The existence of multiple property files matching the same property name will cause a compile-time error.

File format

To ease the NMS-NG upgrade, the NMS-NG property file format remains largely backwards compatible with the current format.

Property files consists of RFC 822-style header fields, including support for line continuations.[1] As a context-free grammar, the syntax is:[2]

```
property_file = field*
field         = name ':' value /\n|$/
name          = /[a-zA-Z$]+/
value         = /([^\n]|\n )*/
```

Additionally:

- Each field may impose additional syntax constraints on its value.
- Newline characters in values are removed as per RFC 822.[3]
- Only ASCII characters have been observed in the existing property files, but for good measure, the encoding of property files shall be considered UTF-8.
- Both Windows-style (CRLF) and Unix-style (LF) newlines are acceptable, and both shall match `'\n'` in the context-free grammar above.

Eight different field names are recognized, all but one belonging to one of four functional groups:

`$Id`	ignored (VCS tag)
`datatype`	value constraint
`values`	value constraint
`mode`	availability constraint
`unittype`	availability constraint
`unit`	descriptive text
`description`	descriptive text
`replaces`	alias definitions

- Value constraints limit the possible values assigned to the property.
- Availability constraints limit the situations in which the property may be used at all.
- Descriptive text provide user guidance for correct property usage.

[1] [RFC822], section 3.1.1. "For convenience, the field-body portion of this conceptual entity can be split into a multiple-line representation; this is called 'folding'. The general rule is that wherever there may be linear-white-space (*not* simply LWSP-chars), a CRLF immediately followed by *at least* one LWSP-char may instead be inserted."

[2] This notation for context-free grammars is discussed in section 4.5.

[3] *ibid.* "Unfolding is accomplished by regarding CRLF immediately followed by a LWSP-char as equivalent to the LWSP-char."

```
unittype: AP,TU
datatype: list<string>
values:
unit:
description: The Service Set IDentifier is the network name
  of the wireless LAN that will be used. On TU several SSIDs
  are allowed.
mode: TGMT,CBTC,PDS
```

Figure 10: *The* `net.wlan.ssid` *property file.*

- Alias definitions give alternative names to a property.

Besides the information listed in the property files, properties are classified as either *internal* or *external*, depending on the location of the property file in one of two directories. Internal properties are reserved for use by advanced users and debugging, and are e.g. hidden by default by the Airlink `var` tool.

An example of an NMS-NG property file can be seen in figure 10.

Default values

The current Airlink property files define default values for properties, to be used if no value is specified for a given node and property. NMS-NG, on the other hand, has no explicit support for default values.

In NMS-NG, default values should be provided by a common base class (e.g. `Defaults`), from which all nodes inherit (directly or indirectly). This class would usually be defined in a separate `defaults.conf` file that may be used unchanged across Airlink deployments.

Some of the existing Airlink property files also specify separate `defaultTGMT`, `defaultCBTC` and `defaultPDS` fields, for specifying defaults that depend on the `sys.mode` property (which is set to either `TGMT`, `CBTC` or `PDS`). In NMS-NG, this should be implemented by providing separate `TGMTDefaults`, `CBTCDefaults` and `PDSDefaults` classes (all of which should likely inherit from a common `Defaults` class).

datatype

The following datatypes (field `datatype`) have been observed in the existing Airlink property files:

`string`:
: an arbitrary text string.

`int`:
: a signed 32-bit integer quantity (a C signed int).

`unsigned`, `unsigned int`:
: an unsigned 32-bit integer quantity (a C unsigned int).

`bool`:
: a boolean value (either `true` or `false`).

`float`:
: an IEEE 754 single precision (32-bit) floating point value.

`IPv4Address`:
: an IPv4 address in dotted-decimal notation.

`IPv4AddressNet`:
: an IPv4 address and subnet mask in CIDR notation (`1.2.3.4/5`).

`list<string>`, `list<unsigned>`, `list<IPv4Address>`:
: a space-separated list of values.

The current use of datatypes is somewhat haphazard. E.g. `boot.gateway.ip` is declared a string, rather than an `IPv4Address`, and `crypt.ipsec.debug.types` (also declared a `string`) contains a *comma*-separated list (or set, really) of strings.

Based on a careful review of the 279 existing property files, NMS-NG implements the following datatypes for improved validation and tool support:

`string`:
: an arbitrary text string.

`int`:
: a signed 32-bit integer quantity (a C signed int).

`bool`:
: a boolean value (either `true` or `false`).

`float`:
: an IEEE 754 single precision (32-bit) floating point value.

`list<T>`:
: a space-separated list of elements of type `T`.

`set<T>`:
: like `list<T>`, but unordered and without duplicates.

`IPv4Address`:
: an IPv4 address in dotted-decimal notation.

`IPv4AddressNet`:
: an IPv4 address and subnet mask in CIDR notation (`1.2.3.4/5`).

`IPv4AddressPort`:
: an IPv4 address and port number, separated by a colon (`1.2.3.4:5555`).

`Password`:
: a password hash in standard Unix `crypt` format.

For this to work, the following issues must be resolved:

- `unsigned` properties become `int` properties, each with a separate additional value constraint that the value must be non-negative (described below).
- Use of existing comma-separated properties is audited, and the properties changed to space-separated. Since much of the Airlink code is in the form of shell-scripts, settling on space-separated lists seems appropriate.

values

The following value constraints (in the field `values`) have been observed in the existing Airlink property files:

- blank (no constraint).
- comma-separated set of permitted values (very common for `bool` properties, where the information is rather redundant, but also used for more informative purposes).
- space-separated set of permitted values.
- the string `numbers` (redundantly specified for an `int` property).
- the string `0-?` indicating an (open-ended) range of permitted values (used redundantly for an `unsigned` property, but useful if the `unsigned` datatype is removed).
- the string `1-` indicating an (open-ended) range of permitted values.
- a string such as `1-10000` indicating a range of permitted values.
- the string `space separated list of allowed ssid's`, carrying information that should rightly be placed in the `datatype` field (which should be `list<string>` instead of `string`) and the `description` field.
- the string `comma separated list of multicast groups`.
- the string `alivewatch,x,duplicate,y`, indicating a pattern to be followed, with `x` and `y` being stand-ins for arbitrary numbers.
- the string `line,section,ssid[,...] line,section,ssid[,...]`, indicating a pattern to be followed for each space-separated element.
- the string `2`, indicating only one permitted value (which is also the default for the two properties).
- the string `.`, with no discernible meaning.

NMS-NG supports the following types of value constraints:

- comma separated list of permitted values: `foo,bar,baz`
- regular expressions: `/expr/`
- numeric ranges: `10..24`
  - the use of `-` (hyphen-minus) is avoided, to avoid confusion with signed values.
  - open-ended ranges: `10..`
  - open-ended ranges: `..12`

For `list<T>` and `set<T>` types, the constraints apply to each separate element.

mode, unittype

The `mode` field must contain a comma-separated list of modes (e.g. `TGMT` or `CBTC`), or the special string `ALL`. If not set to `ALL`, the property will only be available for nodes that have `sys.mode` set to one of the listed modes.

Similarly, the `unittype` field must contain a comma-separated list of node types (e.g. `AP` or `TU`), or the special string `ALL`. If not set to `ALL`, the property will only be available for nodes that have `boot.system` set to one of the listed node types.

NMS-NG will emit a warning if a property is assigned in a class or node definition, and that definition either defines or inherits a `sys.mode` or `boot.system` value that renders the property unavailable.

unit, description

The `unit` and `description` fields contain descriptive text, which is only intended for human consumption.

replaces

The `replaces` field (new in NMS-NG) is a comma-separated list of property names, which are rendered obsolete by the current property. The listed names become deprecated aliases for the current property, but will continue to work, yielding a warning on every use.

This feature is used for properties that change their name. If instead a property is deprecated, but no replacement property is available, the deprecated property should simply be deleted. The configuration tools will yield a warning when an undefined property is used.

3.4 Configuration files

An NMS-NG configuration consists of a set of files, each of which is either a *primary* or a *secondary* configuration file.

Primary configuration files are parsed by the NMS-NG compiler and define configuration classes, nodes and security policy. They exist in two formats, *linear* (source code style) and *tabular* (CSV files).

Secondary configuration files are opaque binary files (e.g. cryptographic keys or firmware), which are referenced by filename by the primary configuration files, and never actually parsed by the NMS-NG compiler.

A class or node definition consists of an identifier (its *name*) and any number of property value assignments. Class and node names must obey the standard C rules for identifiers, whereas the rules for property names are more lax, as they can contain periods.

Assigning the same property twice for the same class or node is an error, as is multiple class definitions with the same class name.

Multiple *node* definitions with the same name are allowed (as long as there is no overlap in the properties assigned in each node definition); the definitions are then simply merged. Splitting node definitions across multiple files is useful because different files may be associated with different privileges (discussed in section 3.5).

Linear configuration files

Using a syntax inspired by C and shell scripts, the linear configuration files define classes and security policy (the `grant` keyword, discussed in section 3.5).

Property values can be simple literal values, or compile-time expressions. Properties may also obtain their value from external files (secondary configuration files).

The detailed syntax is shown as a context-free grammar in figure 11. To keep things simple, the shown CFG does not account for comments and white-space. (Comments start with a `#`, and run until the end of the line. White-space is ignored everywhere except in quoted strings, as one would expect.)

The details of configuration file parsing are discussed at length in section 4.

A simple example

This example defines a class `MyNode`, inheriting from the `Defaults` class. Properties `sys.mode` and `nms.ip` are set to the literal strings "TGMT" and "192.168.1.1", while `snmp.ip` is set to the compile-time expression `nms.ip`. Hence, if a node inheriting from `MyNode` overrides the `nms.ip` value, it will effectively set both `nms.ip` and `snmp.ip`.

```
config           = (class | grant)*
class            = "class" identifier classBase? '{' property* '}'
classBase        = '(' identifier (',' identifier)* ')'
property         = property_ref '=' property_value
property_ref     = ( identifier | compiletime_expr )
                   ( '.' ( symbol | compiletime_expr ) )*
property_value   = quoted_value | unquoted_value | file_value
unquoted_value   = symbol | compiletime_expr
quoted_value     = '"' ( /[^{"\\]+/ | /\\./ | compiletime_expr )* '"'
file_value       = '@"' ( /[^{"\\]+/ | /\\./ )* '"'
                   ( '[' sha1_hash ']' )?

compiletime_expr = '{' add_expr '}'

add_expr         = mult_expr ( ( '+' | '-' ) mult_expr )*
mult_expr        = basic_expr ( ( '*' | '/' | '//' | '%' )
                                basic_expr )*
basic_expr       = property_ref | literal | '(' add_expr ')'

grant            = "grant" privilege "to" principal
principal        = sha1_hash
privilege        = "set-prop" '(' property_glob ')' |
                   "set-all-prop" | "set-all-ext-prop" |
                   "inherit" '(' identifier ')' |
                   "inherit-all" | "define-node"
property_glob    = ( identifier | '*' ) ( '.' ( symbol | '*' ) )*

literal          = integer | quoted_string
quoted_string    = '\'' ( /[^'\\]+/ | /\\./ )* '\''
integer          = /-?[0-9]+/
identifier       = /[_a-zA-Z][_a-zA-Z0-9]*/
sha1_hash        = /[0-9a-fA-F]{40}/
symbol           = /[_a-zA-Z0-9]+/
```

Figure 11: *Syntax of linear configuration files (using the notation described in section 4.5).*

```
class MyNode(Defaults) {
    sys.mode = TGMT
    nms.ip = "192.168.1.1"
    snmp.ip = {nms.ip}
}
```

Complex expressions
Besides simple property references, NMS-NG supports a simple expression language. The syntax as currently implemented is given in figure 11. The goal is to remove needless call-outs to external tools for simple string processing, but further work is required to determine the exact features to be supported.

In the following example, the node number is passed through the `%` formatting operator (as implemented in the Python language, and similar to C's `sprintf`) to zero-extend the number to three digits. The resulting `name` is `AP012`.

```
boot.system = "AP"
node.no = 12
name = "{boot.system}{'%03d' % node.no}"
```

Expressions in property references
Compile-time expressions are not limited to property values, but can also be used in property *names*. This feature is particularly useful for wildcard properties, but not limited to these.

The following example assigns the value "10.16.0.1" to the property `net.eth0.ip`. One could easily imagine these three assignments being split between three diffent configuration files, as suggested by the comments.

```
net.{boot_ifc}.ip = {boot_ip}   # Airlink default
boot_ifc = eth0                 # Firmware specific setting
boot_ip = "10.16.0.1"           # Deployment specific setting
```

Expressions in property names must evaluate to a symbol (`/[_a-zA-Z0-9]+/`); in particular, the expression value may not contain a period (`'.'`).

Referencing external files
Property values may be loaded from external files. Filenames are relative to the location of the input configuration file.

```
firmware = @"firmware/axpl52.bin"
initscript = @"ap.d/"
```

As the second line shows, entire directories may be referenced, in which case each file in that directory will be concatenated, and the result assigned to the property.

```
node,class,unitno,location.desc
CSR01,CSR,1,Equipment room
CSR02,CSR,2,Equipment room
AP01,AP,1,"Trackside, upside"
AP02,AP,2,"Trackside, downside"
TU01,TU,1,Train X
TU02,TU,2,Train Y
```

Figure 12: *Example of defining nodes in a tabular configuration file.*

```
class,location.desc,location.utm
EquipmentRoom,Equipment room,17T 630084 4833438
TracksideUp,"Trackside, upside",17T 630123 4833793
TracksideDown,"Trackside, downside",17T 630197 4833802
```

Figure 13: *Tabular class definitions for specifying the locations of nodes.*

Tabular configuration files

Both nodes and classes can be defined using a CSV (comma-separated values) file. The CSV files must start with a header row, followed by zero or more data rows.

For compatibility with localized versions of Microsoft Excel, NMS-NG transparently handles SSV (semicolon-separated values) files as well.

Nodes

A node configuration CSV file must start with a column named `node`, specifying node names, one or more columns named `class`, specifying names of classes to inherit from, and finally one or more columns named after properties, specifying literal values to assign to these properties.

If a property column cell is empty, the property will not be assigned for that node.

Classes

Classes can also be defined using CSV files, if a large number of heterogeneous classes must be created and the linear configuration file format is not appropriate. CSV files do not support expressions or external file references, though.

A class configuration CSV file must start with a column named `class`, specifying class names, zero or more columns named `superclass`, specifying names of classes to inherit from, and finally one or more columns named after properties, specifying literal values to assign to these properties.

If a property column cell is empty, the property will not be assigned for that class.

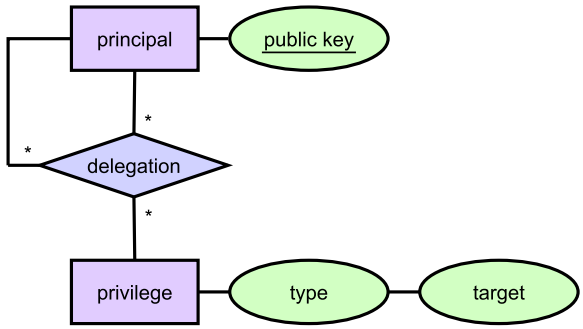


Figure 14: *The underlying data model of the NMS-NG trust model.*

3.5 Security model

As discussed in the requirements analysis (section 2.3), NMS-NG concerns itself primarily with preserving the configuration integrity.

The security model is specifically designed to support the workflows of each user segment (development, sales and support, and customers, as defined in section 2.1).

Signed configuration files

Configuration files may be cryptographically signed by adding the appropriate OpenPGP ASCII armor [RFC4880] to the configuration file (figure 15).

Signing a configuration file endows the file with the privileges of the security principal who does the signing. All configuration files must be signed before deployment.

In linear configuration files, a SHA-1 annotation is added to every external file reference, to ensure the signature also protects the integrity of the referenced secondary configuration files (figure 16).

Trust model and privilege delegation

NMS-NG implements a trust model inspired by the SPKI (Simple Public-key Infrastructure, [RFC2693]) trust model, though further simplified (figure 14).

A *security principal* is simply defined as the fingerprint (the SHA-1 hash) of the public key of an assymetric keypair usable for cryptographic signatures (e.g. an RSA or DSA public key).

A principal can hold a number of *privileges*, and can delegate these privileges to other principals by signing a configuration file with delegation instructions.

```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA1

node,class
CSR01,CSR
TU01,TU

-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.11 (GNU/Linux)

iEYEARECAAYFAlAFcTwACgkQJL/wjZlVk3dmRky3OpWFxMf14xs1bW/Fcre
n40Anj5buaNNpUYZcTGtjW+BUvbeUDJd
=kgOd
-----END PGP SIGNATURE-----
```

Figure 15: *Example of a signed tabular configuration file.*

```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA1

class Node {
    tu.vg.timeout = 500
    key = @"my.key" [01ce2162f92e80defa270badbe22e3aa69d9]
}

-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.11 (GNU/Linux)

iJwEAQECAAYFAk+9JnYACgkQ5f7VEac6VAP+OKCgRymp27kzZg24k+p1N2
7kZykJmF55L44GK57TI=
=6cY+
-----END PGP SIGNATURE-----
```

Figure 16: *Example of a signed linear configuration file with a SHA-1 annotation.*

Hardcoded into the Airlink system is the *root principal* (i.e. the *root public key*), which has every privilege.

The trust model is purely additive and does not support revocations; once granted, privileges cannot be rescinded. The scale and centralized nature of Airlink deployments makes it quite feasible to simply change the root principal key (thus instantly voiding all existing privileges), if the need arises.

Delegation instructions are placed in linear configuration files, and look like this:

```
grant set-all-prop to 3ce764d4faff5fa810b203bfadf516d38421f11c
```

The above example grants the `set-all-prop` privilege to the principal with the specified public key fingerprint.

Available privileges

`set-prop(`*`prop`*`)`:
: The permission to assign values to the given property. A `'*'` wildcard can replace a dotted element, to match multiple property names.

`set-all-prop`:
: The permission to assign values to all properties.

`set-all-ext-prop`:
: The permission to assign values to all *external* properties.

`inherit(`*`class`*`)`:
: The permission to have nodes and classes inherit from the given class. Security principals are automatically granted permission to inherit from classes defined by themselves.

`inherit-all`:
: The permission to have nodes and classes inherit from all available classes.

`define-node`:
: The permission to define new nodes.

3.6 Workflow support

We now have the building blocks for constructing a secure workflow that satisfy the project requirements. An example setup is given in figure 17.

The Airlink development group controls the root key, but good security practice dictates that most work is done with a subordinate key (here "Airlink development").

With the root key, the development group defines a policy file that delegates all privileges to the "Airlink development" principal, and a subset of privileges to the "Sales and support" principal.

This setup allows for the root key to be easily updated, without having to resign every configuration file (only an updated policy file needs to be signed).

The development group defines a number of default classes (`defaults.conf`), classes specific to a certain mode of operation (`cbtc.conf` in this example), as well as firmware-specific classes (`firmware.conf`), which (besides firmware-specific settings) also include the requisite firmware files (`axpl52.bin` in this example).

The "Sales and support" principal creates a customer-specific configuration file, and delegates just a few privileges to the customer.

They never change the generic configuration files received from the development group, in order to ensure traceability and (of course) because they don't have the signing key.

When sales and support receives updated configuration files from the development group, they simply replace the previously received set with the new files, tweak the customer-specific configuration if required and run regression tests, before forwarding the updated configuration to the customer.

The customer maintains a small configuration file (`nodes.csv`) defining e.g. which train units are located in which trains.

The customer uses the customer interface (web interface), which handles complexities such as signing of configuration files automatically.

Configuration updates from Siemens can be applied immediately, without interfering with modifications made to `nodes.csv`.

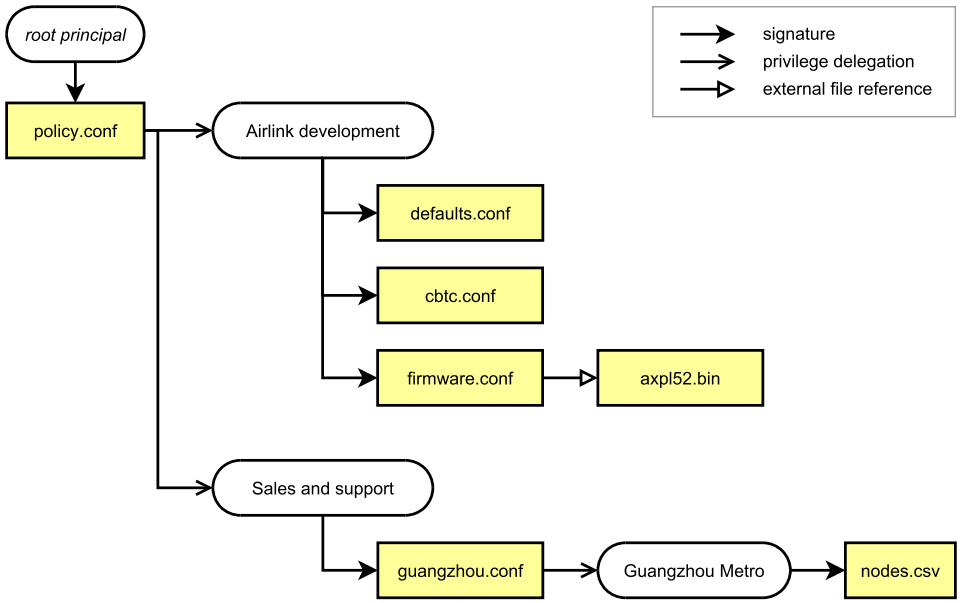


Figure 17: *An example of a complete NMS-NG setup.*

3.7 Implementation language and dependencies

As has been mentioned previously, the current NMS is implemented using an ungraceful mix of languages and technologies (including PHP and Java) that are not used anywhere else in the Airlink system. (Especially the Java dependency is a thorn in the side, adding over a hundred megabytes to the install image.)

For NMS-NG, the Airlink development group wishes for this large and complex set of dependencies to be minimized, with one working paper even calling for everything to be written as plain shell scripts.

As a result, NMS-NG is implemented in the Python language. Python is a modern, high-level language, with superb capabilities in web development, more flexible than Java, but stricter than PHP, with excellent OS APIs (like shell scripts and unlike Java), while providing an OS independent platform. As a bonus, it is a system component in Ubuntu Linux (and most other Linux distributions), one of the primary target platforms.

By design, NMS-NG does not implement its own cryptography, instead requiring GnuPG to be installed on the local system. (Non-cryptography related functionality works without GnuPG, of course.)

NMS-NG has no other dependencies. In particular, it does not depend on a database server (unlike the current NMS).

3.8 Command-line interface

NMS-NG includes a command-line tool (`nms`), which supports manipulation and inspection of a local set of configuration files, as well as communication with a CSR.

Signing

Configuration files may be signed using the `nms sign` command:

```
$ nms sign system.conf
```

The command removes any existing OpenPGP ASCII armor, adds correct SHA-1 hashes for all external files, and finally signs the file (adding up-to-date ASCII armor) using GnuPG (`gpg --clearsign`).

`nms unsign` removes the ASCII armor and file hashes again:

```
$ nms unsign system.conf
```

File lists

The `nms` tool can produce a list of all the files making up a given configuration (including secondary configuration files):

```
$ nms files
```

By specifying filenames on the command-line, the tool will only list the specified files and the external files they reference. As such, the above command is equivalent to:

```
$ nms files *.conf *.csv
```

The command can e.g. be used to easily bundle up an entire configuration:

```
$ tar cf backup.tar $(nms files)
```

In verbose mode (`nms files -v`), the command shows the signing status (unsigned, valid or invalid signature) of each file.

Properties

The tool can be used to inspect individual property values for classes and nodes:

```
$ nms var AP retrieswlan0
2
```

Or all properties may be shown for a given node:

```
$ nms var AP001
boot.system=AP
name=AP001
ip=10.141.14.1
retrieswlan0=3
```

This only shows *external* properties, unless in verbose mode(`-v`).

Nodes

Nodes (and selected properties) can be listed in a tabular layout:

```
$ nms nodes ip
node   ip
-----  ------------
CSR01  192.168.64.1
CSR02  192.168.64.2
AP001  10.141.14.1
AP002  10.141.14.2
```

The command `nms nodes --csv` can be used to export the table as a CSV file.

Classes

The tool can list all available classes:

```
$ nms classes
AP
AP1
CSR
Node
TU
```

The `--tree` option causes the classes to be listed as a tree (multiple inheritance may cause the same class to appear multiple times):

```
$ nms classes --tree
o Node
|-o AP
| |-- AP1
|-- CSR
\-- TU
```

Validation

Errors in the configuration may cause the tool to fail with a compile-time error. However, not all commands require a completely valid configuration to succeed; the `nms files` command for instance only requires that the files are syntactically valid, and does not verify e.g. that the class inheritance graph is even valid.

The `nms validate` command request an explicit and complete validation of the configuration, including checking that every file is correctly signed, and that the security policy is obeyed.

Deployment

The configuration may be deployed to a CSR:

```
$ nms deploy 1.2.3.4
```

Deployment to the CSR happens using the `rsync` tool to copy all configuration files over. The rsync protocol provides automatic delta-compression, limiting how much data needs to be copied.

The configuration is validated by `nms` before deployment, and again by the CSR.

Retrieving the active configuration

If no local copy exists, the currently active configuration may be retrieved from the CSR. The configuration files are placed in a specified directory.

```
$ nms retrieve 1.2.3.4 target-dir/
```

Like deployment, configuration retrieval is performed using the `rsync` tool.

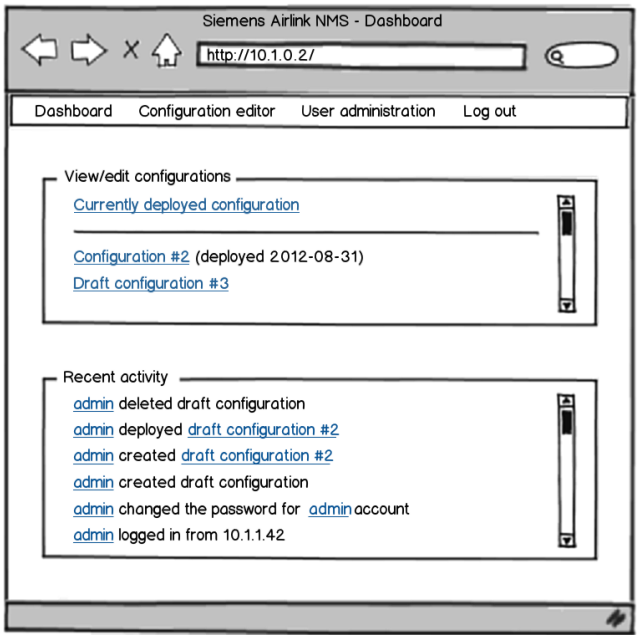


Figure 18: *The dashboard provides a configuration overview and an activity feed.*

3.9 Web interface

The NMS-NG web interface (the "WUI") provides a subset of the features of the command-line interface, and some WUI specific features:

- Deploying a new configuration to a CSR
- Retrieving the currently active configuration from a CSR
- Configuration import/export
- Linear configuration file viewer
- Tabular configuration file editor
- User account management

The goal is to support the daily operations of the customer, and to allow use by non-technical users.

Dashboard

Shown after log in, the dashboard provides shortcuts to fetch and view the currently deployed configuration, and to edit the draft configurations. At the bottom is an activity feed, listing the most recent entries of the WUI log file and ensuring that the user is aware of recent actions taken by other users.

User accounts

A WUI user account corresponds directly to an NMS-NG security principal, though not all security principals have a user account.

The WUI user account database consists simply of a set of files, one per account. Each file contains an encrypted OpenPGP private key that can be used for signing configuration files.

To log in to the WUI, the user must provide a username and passphrase. The username corresponds to the filename of a user account key file, and the passphrase is used to decrypt the key file.

The user is hence granted access to the WUI only upon successfully decryption of a user account key.

This also means that a compromise of the web server does not immediately lead to a compromise of the remaining Airlink system, since all the keys (which are required to actually do anything) are encrypted. (Of course, once a user logs in to the compromised web server, the attacker will be able to impersonate that particular user, but that is a fundamental problem of any web interface.)

The WUI provides a page for basic user account administration (changing passwords, adding and deleting accounts).

To add a new account, the user must upload a key file. A valid key file can be generated by the customer using any compliant OpenPGP implementation (such as GnuPG), or one can be provided by Siemens upon request. (This slight difficulty in creating new accounts was deemed acceptable, since most Airlink deployments will have perhaps two or three user accounts, which rarely change.)

Since each user account is simply a standard OpenPGP key file, technically inclined users can also perform user administration directly on the server.

Configuration management

The WUI can track multiple, separate *draft configurations*, which it can import and export in the form of zip-files. The WUI provides a page for viewing the files of a given configuration, as well as an editor for tabular configuration files.

More than a simple text grid, the tabular configuration file editor provides property hint texts and combo-box (auto-completion) features to assist the user in choosing the right properties and property values.

On request, the WUI retrieves the currently active configuration from a CSR, for viewing by the user. If the user attempts to *edit* the current configuration, it is automatically copied to a draft configuration, which the user can then edit and deploy.

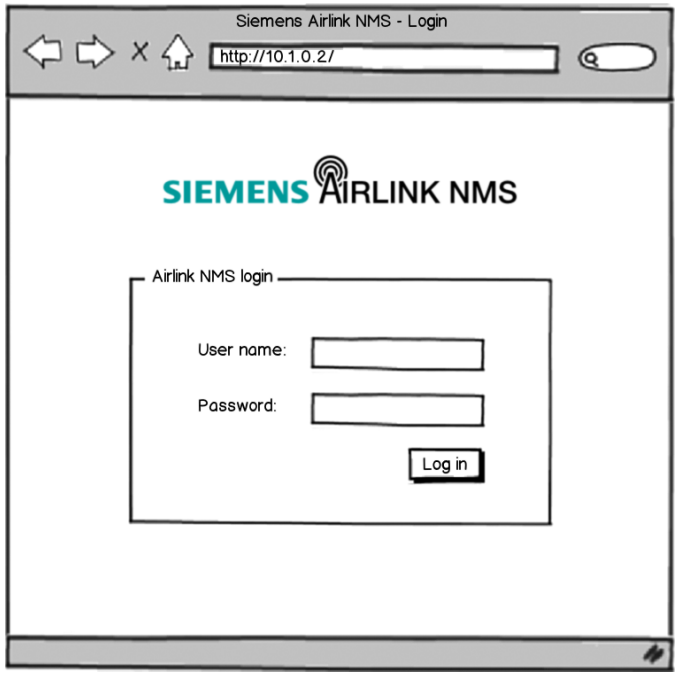


Figure 19: *The log in screen.*

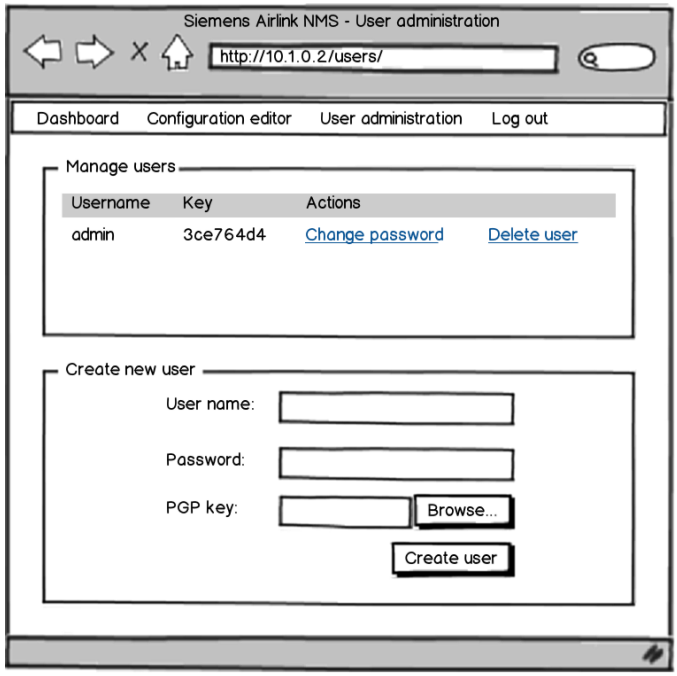


Figure 20: *The user account management screen.*

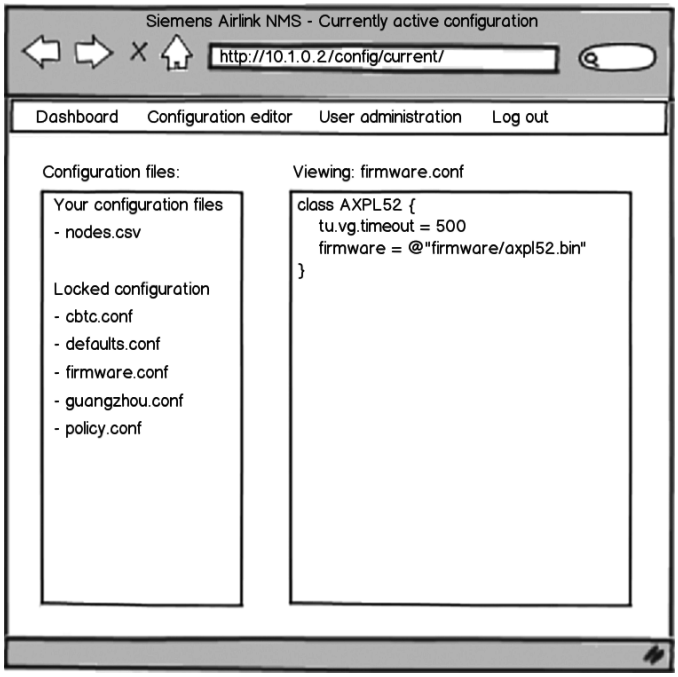


Figure 21: *Viewing a file from the current configuration.*

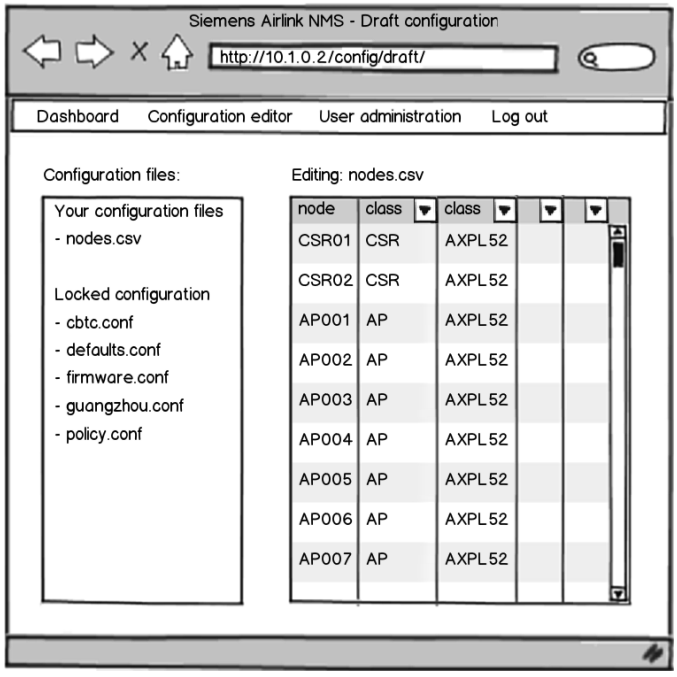


Figure 22: *Editing a file from a draft configuration.*

Step	Chomsky type	Analytic complexity
Lexical	Type 3	Regular
Syntactic	Type 2	Context-free
Semantic	Type 1 or 0	Context-sensitive or higher

Figure 23: *The three levels of analytic complexity correspond loosely to increasing levels in the Chomsky hierarchy of formal language complexity.*

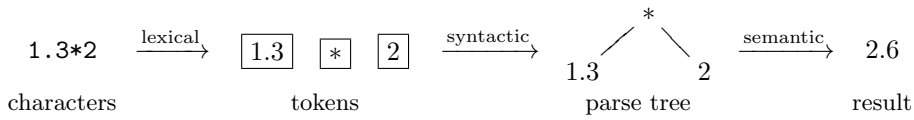


Figure 24: *How a simple calculator might parse user input.*

4 Implementing a parser framework

4.1 Lexical, syntactic and semantic analysis

The parsing of formal languages (programming languages in particular) is typically divided into three steps, in order of increasing complexity (figure 23).

Lexical analysis converts a sequence of characters into a sequence of tokens by repeated application of a regular language recognizer (that is, using regular expressions). Tokens are typically words, literals, operators and punctuation. "Null tokens" (usually whitespace and comments) are discarded during this step.

Syntactic analysis converts the sequence of tokens into a parse tree, according to a context-free grammar.

Semantic analysis converts the parse tree in a non-formalized and program-specific manner into the desirable end-result.

Figure 24 shows an example of this process.

While this separation is mathematically redundant (since each step has enough analytic complexity to perform the previous steps on its own), it greatly simplifies the parsing process:

- The non-formalized semantic analysis is separated from the syntactic analysis, allowing the well-established formalisms of context-free grammars to be used for the latter.
- The context-free grammar is greatly simplified by not dealing with null tokens, not to mention by not operating at the individual character level.

Lexical analysis of string literals

A side effect of this three-step approach is that all parts of the language where whitespace is significant (such as string literals and one-line comments) must be describable using regular expressions, because the context-free step never sees the whitespace. Hence even the most complex token in the C language, the string literal, can be described by a regular expression, which roughly looks like this:

```
"([^"\\]|\\.)*"
```

The exact definition in the C standard [ISO9899] is equivalent to:

```
L?"([^\n"\\]|\\['"?\\abfnrtv]|\\[0-7]{1,3}|\\x[0-9a-fA-F]+|
    \\u[0-9a-fA-F]{4}|\\U[0-9a-fA-F]{8})*"
```

Slightly more complex, but definitely regular.

This fact holds true for most derivatives of the C language, as well. For instance, the definition of the string literal in the Java Language Specification [JLS:SE7] is equivalent to the following regular expression:

```
"([^\n\r"\\]|\\[btnfr"'\\]|\\[0-7]{1,2}|\\[0-3][0-7][0-7])*"
```

(The \u Unicode escape sequence is absent, because it is handled by an even earlier parsing step.)

Lexical analysis of the NMS-NG linear configuration language

In the NMS-NG linear configuration language, string literals can contain embedded expressions, a feature found in many Unix shell languages (such as the Bourne shell) and derivatives (such as PHP and Perl).

Obviously, the whole string literal including expressions cannot be handled as a single lexical token, since the expression syntax is context-free, not regular. However, the lexical rules inside a string literal are quite different from those on the outside – whitespace is significant, for instance. Thus an NMS-NG string literal cannot be split into separate tokens either, as the analytic complexity of this too is above regular.

The result is that the NMS-NG configuration language cannot be analysed using regular grammars; the syntactic analysis must be performed directly on the raw sequence of characters. To deal with whitespace and comments gracefully, the developed parser framework contains explicit support for "null tokens" in the syntactic analysis step.

4.2 Look-ahead

NMS-NG implements an LL parser (specifically, a recursive descent parser). Of LL parsers, LL(1) are the most common. This means that whenever the parser encounters a production with multiple alternatives, it makes its choice based only on the next input token (a look-ahead of 1).

LL(1) parsers perform better than parsers with higher look-ahead, and also benefit from more appropriate error reporting in case of syntax errors.

As an example of the latter, take the input string `user;` and the following grammar:

```
entity = user | group
user   = "user" identifier ';'
group  = "group" identifier ';'
```

The lexical analysis yields two tokens, `user` and `;`.

An LL(1) parser will start at `entity`, then (using one token of look-ahead) choose the `user` production over `group`. After consuming the `user` token, the parser will expect to find an `identifier`, but instead finds a `;`, provoking an error of the form "Expected identifier, found ';' at input position 4".

An LL parser with higher look-ahead will start at `entity`, but then deduce that neither alternative is a valid match for the input, provoking an error of the form "Expected entity, found 'user;' at input position 0", a much less helpful message.

Solution

As explained in section 4.1, since the NMS-NG parser has no separate lexical step, it technically operates with tokens corresponding to individual characters.

Having a look-ahead of only one character is hardly enough for most languages (Java would e.g. not be able to distinguish the keywords "public" and "private"), so clearly an LL(1) parser is not an option in our case. Instead, the NMS-NG parser has a variable amount of look-ahead, LL(*), depending on the specific grammar production currently being matched.

In practice, the look-ahead is typically implemented to be a single keyword, identifier, operator or punctuation character. The end result is that the parser resembles a traditional LL(1) parser with a preceding lexical analysis step, except that it handles the aforementioned problem of strings with embedded expressions.

4.3 Left-recursion

Due to their simplistic nature, LL parsers with finite look-ahead have trouble with a number of context-free grammar constructs. The most common problem is left-recursion, which frequently occurs in grammars that avoid Kleene repetition.

Take for example this simple expression syntax, which only supports subtraction:

```
expr = number | expr '-' number
```

The LL parser has no way to distinguish these alternatives, since both start with a `number` token (either directly or recursively).

A common work-around is to replace left-recursion with right-recursion:

```
expr = number ( '-' expr | )
```

This grammar is weakly equivalent to the left-recursive one above, in that it matches the same expression language. However, it is not strongly equivalent since it produces a right-associative parse tree instead of a left-associative tree. This causes problems in the semantic analysis (since the subtraction operator is supposed to be left-associative), leading to incorrect results (figure 25). There are workarounds, of course, but they needlessly complicate the semantic analysis.

Solution

Most left-recursion problems are solved by adding parser support for the Kleene star, which also allows the grammar to be specified in a natural and concise manner. The resulting parse tree becomes flat, rather than nested (figure 25).

```
expr = number ( '-' number )*
```

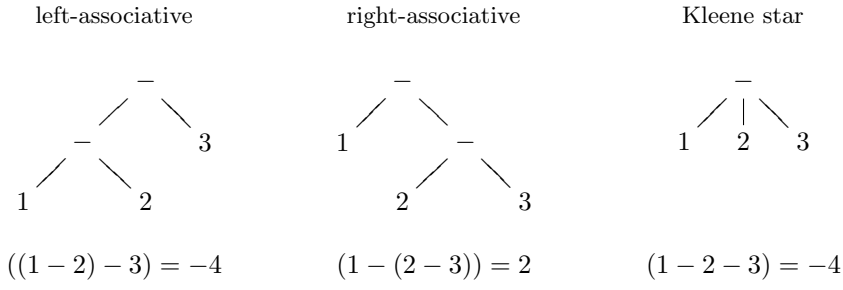


Figure 25: *How associativity affects the result.*

4.4 Input reconstruction

One NMS-NG feature is the ability of the `nms` tool to read a source file, add SHA-1 annotations to file references, and write out the resulting source file otherwise unchanged.

Most parser frameworks struggle to support this task, for at least two reasons:

- Since whitespace and comments are discarded during lexical analysis, they cannot be recovered later.
- They do not retain source-accurate token representations, e.g. discarding information about leading zeros in a decimal literal.

The NMS-NG parser accurately tracks all source elements, including whitespace and comments, to ensure that an accurate reconstruction is possible.

4.5 Parser objects

The most basic object of the parser is the *recognizer*, which is an object that, given an input to recognize, yields either succees or failure. On success, the recognizer *consumes* zero or more characters of input and also returns a *match object* corresponding to these input characters.

A recognizer can be *named* or *anonymous* (inline). Named recognizers permit recognizers to be used more than once, and enables recursive recognizers (in the form of productions that refer to themselves).

An *atomic* recognizer yields likewise atomic match objects, which are the leaf nodes of the *parse tree*.

The NMS-NG parser provides three different atomic recognizers:

- The *literal* recognizer matches a specific character string. NMS-NG uses this to match punctuation (e.g. braces) and operators. A single-quoted string (e.g. `'{'`) denotes a literal recognizer.
- The *word* recognizer also matches a specific character string, but the string must be followed by a non-alphabetical character in the input. NMS-NG uses this to match keywords (e.g. `class`), since using a plain literal recognizer would also match e.g. the word `classic`. A double-quoted string (e.g. `"class"`) denotes a word recognizer.
- The *regular expression* recognizer matches an arbitrary regular expression. NMS-NG uses this to match all other tokens, such as identifiers, literals, comments and whitespace. A string surrounded by slashes (e.g. `/[0-9]+/`) denotes a regular expression recognizer.

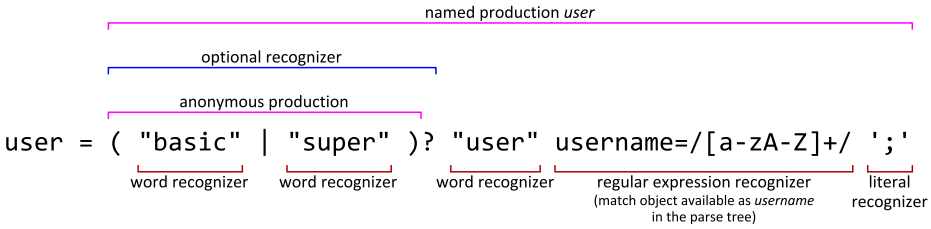


Figure 26: *An annotated production declaration.*

A *complex* recognizer yields match objects that are divisible into submatches (the internal nodes of the parse tree).

The NMS-NG parser provides three different complex recognizers:

- A *production* recognizer matches one of several given alternatives; each alternative being a sequence of subrecognizers. The default implementation uses the first recognizer of each alternative (a look-ahead of one "token") to determine which alternative to pursue.
- A *Kleene star* recognizer repeatedly tries to match the input against a given subrecognizer, yielding a list of submatches. It always succeeds in finding a match, though the match may be empty. The Kleene star recognizer is denoted by an asterisk (`*`) following the target subrecognizer.
- An *optional* recognizer works like the Kleene star, but without repetition. It will hence match zero or one times. The optional recognizer is denoted by a question mark (`?`) following the target subrecognizer.

Null recognizers

Production recognizers may define a set of *null recognizers* for handling whitespace and comments. While parsing an alternative, the production recognizer runs the null recognizers before and after every subrecognizer. As previously explained, the resulting null matches (if any) are not discarded, but included in the parse tree.

Submatch names

Production recognizers may assign names to submatches for the benefit of the semantic analysis step, which can use these names to navigate the parse tree instead of numeric indices. This is particularly important when using null recognizers, as the resulting null matches affect the indices assigned to the other submatches.

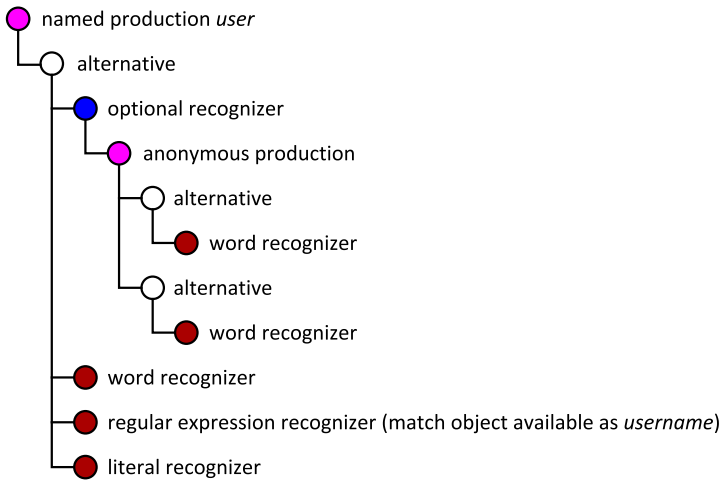


Figure 27: *The parser object tree corresponding to figure 26.*

4.6 A domain-specific language for context-free grammars

Manual construction of the objects of a context-free grammar is a verbose and error-prone task. For this reason, the NMS-NG parser framework defines a small domain-specific language for defining CFGs. This language is of course parsed using the parser framework itself, and resembles the CFG notation used in this report.

```
production  = alternative ( '|' alternative )*
alternative = element*
element     = assignment? recognizer number?
recognizer  = identifier | literal | word | regEx | '(' production ')'
number      = '*' | '?'

assignment  = /([_a-zA-Z][_a-zA-Z0-9]*)\s*=/
identifier  = /[_a-zA-Z][_a-zA-Z0-9]*\b/
literal     = /'([^'\\]|\\.)*'/
word        = /"([^"\\]|\\.)*"/
regEx       = /\/([^/\\]|\\.)*\//
```

By using the reflection capabilities of Python, the NMS-NG parser does away with the separate "compiler compiler" preprocessing step required by many parser frameworks. The result is a very compact notation for declaring a context-free grammar. A full example is given in figure 28.

4.7 Parse tree transforms

For the initial semantic analysis, the NMS-NG parser framework implements a transformation model that operates directly on the parse tree. This enables transformation patterns to be written in a declarative manner, and automatically be applied to matching nodes of the parse tree.

For simple tasks, the entire semantic analysis can be performed this way, with no further processing required. An example can be seen in figure 28, in which `MyTransform` transforms the raw parse tree into the final numerical result.

```
import operator as ops
from libnms.parse.grammar import *
from libnms.parse.smartgrammar import *
from libnms.parse.transform import *


class MyMatch(SmartMatch):
    """ A customized SmartMatch that automatically skips whitespace. """
    class Recognizer(Production):
        nullRecognizers = [ RegularExpression('\s+') ]


class MyGrammar(SmartGrammar):
    class Root(MyMatch):
        """ a=additive /$/ """

    class Additive(MyMatch):
        """ m=multiplicative _=(op=opAdditive m=multiplicative)* """

    class Multiplicative(MyMatch):
        """ p=parenthesis _=(op=opMultiplicative p=parenthesis)* """

    class Parenthesis(MyMatch):
        """ val=integer | '(' val=additive ')' """

    integer          = RegularExpression('-?[0-9]+')
    opAdditive       = RegularExpression('[-+]')
    opMultiplicative = RegularExpression('[*/]')


class MyTransform(MatchTransform):
    root             = lambda s, v: v.a
    additive         = lambda s, v:
        reduce(lambda acc, elm: elm.op(acc, elm.m), v._, v.m)
    multiplicative   = lambda s, v:
        reduce(lambda acc, elm: elm.op(acc, elm.p), v._, v.p)
    parenthesis      = lambda s, v: v.val
    integer          = lambda s, v: int(v._text)
    opAdditive       = lambda s, v: ops.add if v._text == '+' else ops.sub
    opMultiplicative = lambda s, v: ops.mul if v._text == '*' else ops.div


input = "1 + 2 * (3 + 4)"
parseTree = MyGrammar.root.match(SourcePosition(input))  # Syntactic step
output = MyTransform(MyGrammar)(parseTree)               # Semantic step
assert output == 15
```

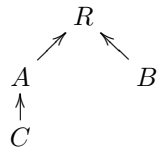
Figure 28: *A simple calculator example using the NMS-NG parser framework.*

5 Implementing multiple inheritance

A fundamental fact of any inheritance system (single inheritance systems, too) is that a class may inherit conflicting definitions from its parents. Though some languages require manual conflict resolution,[4] most resolve conflicts by computing a *class precedence list* (CPL, also known as the *method resolution order*) for each class – and using the first definition found in CPL order.

A *linearization* is an algorithm for computing class precedence lists. There exists many different linearizations; we shall look at three in common use.

5.1 Single inheritance linearization



$$\begin{array}{lcl} \text{bases}(R) & = & \epsilon \\ \text{bases}(A) & = & R \\ \text{bases}(B) & = & R \\ \text{bases}(C) & = & A \end{array} \qquad \begin{array}{lcl} \text{cpl}_1(R) & = & R \\ \text{cpl}_1(A) & = & AR \\ \text{cpl}_1(B) & = & BR \\ \text{cpl}_1(C) & = & CAR \end{array}$$

Figure 29: *A single inheritance class tree, listing CPLs for each class.*

In a single inheritance system, each class has at most one base class.

The CPL of C is trivially defined as the vertices of the path extending from C to the root of the class hierarchy.

5.2 Properties of linearizations

With multiple inheritance, the construction of the CPL becomes more complex, with several different linearizations in common use.

[Barrett96] lists four desirable properties for a linearization:

- In an *acceptable* linearization, only the shape of a class' inheritance graph may be used in determining its CPL. E.g. having class names or the declaration order in the source code affect the linearization is unacceptable. In practice, this property is a given; the author knows of no unacceptable linearizations seeing actual use.
- A *monotonic* algorithm guarantees that every CPL is an extension (without reordering) of the base class CPLs. [Ducournau94]

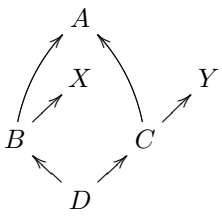
[4] For example, C++ requires manual conflict resolution when a class virtually inherits the same virtual method from multiple base classes.

- A linearization that observes *local precedence order* will not produce a CPL which is inconsistent with the local precedence order (i.e. the bases tuple) of that class, or any superclass. That is, if class A precedes class B in bases(C), the CPLs of C and all of its subclasses should have A before B.
- *Extended precedence graph (EPG) consistency* further constrains CPLs to ensure consistency across the inheritance graph. This property is best explained by an example; see figure 30.

Already, the first two properties are in conflict: There is no acceptable linearization which is monotonic for all conceivable inheritance graphs. [Ducournau94]

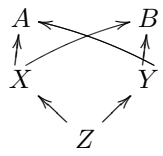
An inheritance graph is said to be *inconsistent* under a given linearization, if the algorithm cannot produce a CPL for every class in the inheritance graph.

It is also easy to see that the local precedence order property on its own causes some inheritance graphs to be rejected as inconsistent (figure 31).



class	bases	cpl_D	cpl_C3
X	ϵ	X	X
Y	ϵ	Y	Y
A	ϵ	A	A
B	AX	BAX	BAX
C	AY	CAY	CAY
D	BC	$DBCAYX$	$DBCAXY$

Figure 30: *An example of EPG inconsistency (from* [Barrett96]*).* cpl_D *is the Dylan linearization, which (while monotonic and observant of local precedence order) is not EPG consistent, and gives* Y *precedence over* X *despite* B *coming before* C*.* cpl_C3*, on the other hand, satisfies all three properties.*



$$\begin{array}{lcl} \text{bases}(A) & = & \epsilon \\ \text{bases}(B) & = & \epsilon \\ \text{bases}(X) & = & AB \\ \text{bases}(Y) & = & BA \\ \text{bases}(Z) & = & XY \end{array} \qquad \begin{array}{lcl} \text{cpl}(A) & = & A \\ \text{cpl}(B) & = & B \\ \text{cpl}(X) & = & XAB \\ \text{cpl}(Y) & = & YBA \\ \text{cpl}(Z) & = & ? \end{array}$$

Figure 31: *An example of an inheritance graph which is inconsistent under any linearization observing local precedence order.*

5.3 Depth-first linearization

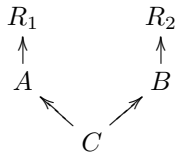
The *depth-first* linearization is a trivial extension of single inheritance linearization:

$$\text{bases}(C) = B_1 \ \cdots \ B_n \quad \Rightarrow \quad \text{cpl}_{\text{df}}(C) = C \ \cdot \ \text{cpl}_{\text{df}}(B_1) \ \cdots \ \text{cpl}_{\text{df}}(B_n)$$

The depth-first algorithm is clearly non-monotonic. If e.g. R_1 and R_2 are collapsed into a single class R in figure 32, we get figure 33. As can be seen, R comes before B in $\text{cpl}_{\text{df}}(C)$, despite B coming before R in $\text{cpl}_{\text{df}}(B)$, which violates monotonicity.

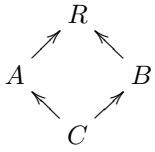
In general, the algorithm provides non-monotonic results when there's more than one path from one class to another.

Despite its flaws, this algorithm is often used due to its simplicity, e.g. in Perl[5] and older versions of Python.[6]



$$\begin{array}{lcl@{\qquad}lcl}
\text{bases}(R_1) & = & \epsilon & \text{cpl}_{\text{df}}(R_1) & = & R_1 \\
\text{bases}(R_2) & = & \epsilon & \text{cpl}_{\text{df}}(R_2) & = & R_2 \\
\text{bases}(A) & = & R_1 & \text{cpl}_{\text{df}}(A) & = & AR_1 \\
\text{bases}(B) & = & R_2 & \text{cpl}_{\text{df}}(B) & = & BR_2 \\
\text{bases}(C) & = & AB & \text{cpl}_{\text{df}}(C) & = & CAR_1BR_2
\end{array}$$

Figure 32: *A multiple inheritance class tree, listing depth-first CPLs.*



$$\begin{array}{lcl@{\qquad}lcl}
\text{bases}(R) & = & \epsilon & \text{cpl}_{\text{df}}(R) & = & R \\
\text{bases}(A) & = & R & \text{cpl}_{\text{df}}(A) & = & AR \\
\text{bases}(B) & = & R & \text{cpl}_{\text{df}}(B) & = & BR \\
\text{bases}(C) & = & AB & \text{cpl}_{\text{df}}(C) & = & CARBR \equiv CARB
\end{array}$$

Figure 33: *The depth-first linearization is not monotonic.*

[5]Depth-first is the default linearization in Perl 5, but not in the upcoming Perl 6.

[6]Depth-first was used for all classes before Python 2.1, for "old-style" classes in versions 2.1 through 2.7, and eliminated in Python 3.

5.4 C3 linearization

The *C3 linearization*, first described by [Barrett96], satisfies all four previously mentioned properties, and is thus in a sense optimal.

Central to the algorithm is the MERGE function, which takes one or more input strings and merges them into a new string that contains each input symbol exactly once, while preserving the relative ordering of the symbols in each input string.

MERGE works by looking at the heads of every input sequence and picking the first of them that can be added to the result sequence without violating the relative orderings of the remaining input strings. Once a class is *satisfied* by appearing in the result, it is removed from all inputs sequences wherein it occurs. This procedure is repeated until all input sequences are empty.

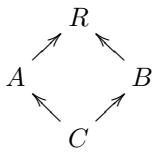
An example of the merge operation is shown in figure 35. The first three columns are input sequences, the fourth column is the result sequence. The horizontal line progresses downwards as the algorithm proceeds; the letters under the line are what remains of the input sequences.

Figure 36 shows how the algorithm deals with the inconsistent inheritance graph from figure 31. After the shown steps, MERGE has to choose between A and B, but both are in the tail of an input sequence, and hence not eligible to go next, causing the algorithm to bail with an error.

C3 determines the CPL by merging the CPL of each base class together with the string of base classes:

$$\begin{gathered}\operatorname{bases}(C) = B_1 \ \cdots \ B_n \\ \Downarrow \\ \operatorname{cpl}_{\mathrm{C3}}(C) = C \ \cdot \ \textsc{merge}\big(\operatorname{cpl}_{\mathrm{C3}}(B_1),\ \ldots,\ \operatorname{cpl}_{\mathrm{C3}}(B_n),\ \operatorname{bases}(C)\big)\end{gathered}$$

The MERGE pseudo-code can be seen in figure 37.



$$\begin{array}{lcl} \operatorname{bases}(R) & = & \epsilon \\ \operatorname{bases}(A) & = & R \\ \operatorname{bases}(B) & = & R \\ \operatorname{bases}(C) & = & AB \end{array} \qquad \begin{array}{lcl} \operatorname{cpl}_{\mathrm{C3}}(R) & = & R \\ \operatorname{cpl}_{\mathrm{C3}}(A) & = & AR \\ \operatorname{cpl}_{\mathrm{C3}}(B) & = & BR \\ \operatorname{cpl}_{\mathrm{C3}}(C) & = & CABR \end{array}$$

Figure 34: *A multiple inheritance class tree, listing C3 CPLs.*

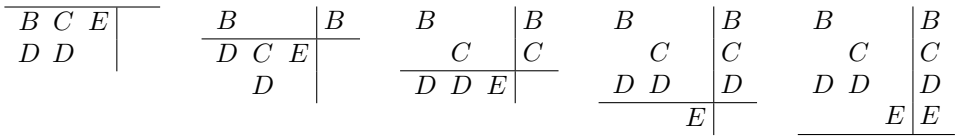


Figure 35: *Example of merging:* MERGE($BD,\ CD,\ E$) = $BCDE$

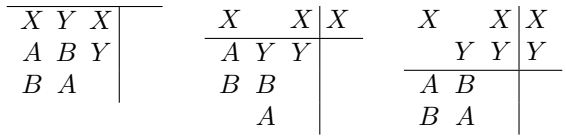


Figure 36: *Example of a failed merge:* MERGE($XAB,\ YBA,\ XY$) = ?

```
MERGE(X_1, ..., X_n)
 1  result ← ϵ
 2  while ∃X_i : X_i ≠ ϵ
 3        do next ← ϵ
 4           ▷ Find the next class to satisfy. next must not be in the tail of any
 5           ▷ sequence, since that would mean another class has priority.
 6           for each X_i where X_i ≠ ϵ
 7                do if ∀X_j : head(X_i) ∉ tail(X_j)
 8                      then next ← head(X_i)
 9                           break
10           if next = ϵ
11              then error Inconsistent input sequences.
12           result ← result · next
13           ▷ Since next is now satisfied, remove it from all inputs.
14           for each X_i where X_i ≠ ϵ
15                do ▷ Only check the heads; we know next is not in any tail.
16                   if head(X_i) = next
17                      then X_i ← tail(X_i)
18  return result
```

Figure 37: *Pseudo-code for the C3* MERGE *function.*

6 Testing

Since NMS-NG is just a component of the larger Airlink system, and to limit the scope of the project, testing has focused on black box unit testing.

To verify test coverage, the excellent `coverage.py` code coverage tool for Python was used (figure 40).

6.1 Unit tests

Unit tests are implemented using the `unittest` module from the Python standard library, which provides a unit testing framework modelled after JUnit.

As such, the implementation is fairly straightforward: NMS-NG consists of a number of Python packages. A `test` module in each package contains a number of test fixtures (classes), one for each of the remaining modules in the package.

6.2 Documentation tests

A problem with `unittest` tests cases are that they can become rather verbose. For this reason, some of the unit tests employ the `doctest` module of the Python standard library, which is great for simple ad-hoc tests and for testing code with side effects (in particular, code that prints to stdout).

`doctest` works by extracting code samples from documentation comments in the code, running them, and verifying their accuracy. As such, `doctest` provide a twist on Knuth's literate programming paradigm: The documentation *is* the unit tests.

6.3 Generated tests

The tests for the C3 implementation exploit the fact that Python uses C3 internally in its own class model.

The test suite therefore constructs identical inheritance graphs using NMS-NG classes and Python classes, and compares the resulting class precedence lists. The test suite includes a set of manually created test cases, but it also generates 500 random test cases at runtime, with highly complex inheritance graphs, and verifies those. To ensure any errors are reproducible, the random generator is run with a fixed seed, so the same test cases are generated every time.

```
def testGlobs(self):
    pdc = PropertyDefinitionCollection()
    pdc.add(PropertyDefinition('net._.ip'))
    pdc.add(PropertyDefinition('foo.bar._'))
    pdc.add(PropertyDefinition('foo._.baz'))

    self.assertEquals(pdc['net.eth0.ip'], pdc['net._.ip'])
    self.assertEquals(pdc['foo.bar.42'], pdc['foo.bar._'])
    self.assertEquals(pdc['foo.42.baz'], pdc['foo._.baz'])
    self.assertRaises(PropertyDefinitionError,
        pdc.__getitem__, 'foo.bar.baz')
```

Figure 38: *An example of a test case, in this case testing the property definition wildcard feature.*

```
def normalizeLineEndingsUnix(text):
    r""" Normalizes line endings to Unix style.

        >>> normalizeLineEndingsUnix('Hi.\r\n')
        'Hi.\n'
    """
    return text.replace('\r\n', '\n')
```

Figure 39: *A simple utility function, complete with documentation and unit test.*

Coverage report: 82%

Module	*statements*	*missing*	*excluded*	*coverage*
libnms__init__	0	0	0	100%
libnms\conffile__init__	0	0	0	100%
libnms\conffile\parse	127	28	0	78%
libnms\conffile\test	18	0	0	100%
libnms\configurationfile	59	30	0	49%
libnms\confmodel	96	17	0	82%
libnms\datatypes	13	2	0	85%
libnms\parse__init__	0	0	0	100%
libnms\parse\grammar	260	26	0	90%
libnms\parse\smartgrammar	119	18	0	85%
libnms\parse\synth	36	29	0	19%
libnms\parse\transform	66	4	0	94%
libnms\parse\unparse	26	21	0	19%
libnms\propfile__init__	0	0	0	100%
libnms\propfile\model	94	22	0	77%
libnms\propfile\parse	45	16	0	64%
libnms\propfile\test	49	0	0	100%
libnms\util__init__	0	0	0	100%
libnms\util\c3	28	0	0	100%
libnms\util\english	10	0	0	100%
libnms\util\errorhandling	23	0	0	100%
libnms\util\gpg	28	2	0	93%
libnms\util\misc	17	7	0	59%
libnms\util\polymorphic	12	0	0	100%
libnms\util\regexquote	6	0	0	100%
libnms\util\test	93	0	0	100%
Total	**1225**	**222**	**0**	**82%**

coverage.py v3.5.2

Figure 40: *A code coverage report for the NMS-NG project.*

7 Conclusion

The NMS-NG presented in this report is a highly flexible configuration system, and solves numerous problems that the Airlink developers has had with the existing NMS. The accompanying security model improves security significantly, and is carefully designed to not get in the way of the daily workflows of the different Airlink user segments.

The design of NMS-NG is almost complete, with a few details left to be ironed out together with the Airlink development group. The result should provide an effective platform for coming generations of the Airlink system.

The implementation is perhaps 75 % complete. While the core functionality (such as the configuration compiler) is fully functional, there are gaps in the implementation, particularly in the error handling (which is highly rudimentary) and the web interface (which currently only exists as design sketches).

The currently implemented unit tests cover a healthy amount of the application code, but need to be expanded to cover the entire code base. Integration testing (testing NMS-NG together with the remaining Airlink system) has not begun. Once the web interface is working, it will of course also need testing; as with all user interface tests, this can be somewhat tricky. Fortunately, there exist some excellent Python libraries for this purpose, which integrate nicely with the existing `unittest` and `coverage.py` infrastructure.

A References

[ISO9899] *ISO/IEC 9899:TC2 Programming languages – C (committee draft)*. ISO/IEC, 2005.

[Siemens12] *References: Fully automated metro lines worldwide*. Siemens, 2012.

[Barrett96] Barrett, Cassels, Haahr, Moon, Playford, Withington. *A monotonic superclass linearization for Dylan*. In *Proceedings of the 11th ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications*, OOPSLA '96, pp. 69–82. ACM, 1996.

[RFC4880] Callas, Donnerhacke, Finney, Shaw, Thayer. *OpenPGP Message Format*. IETF, 2007.

[RFC822] David H. Crocker. *Standard for the format of ARPA Internet text messages*. IETF, 1982.

[Ducournau94] Ducournau, Habib, Huchard, Mugnier. *Proposal for a monotonic multiple inheritance linearization*. In *Proceedings of the ninth annual conference on object-oriented programming systems, language, and applications*, OOPSLA '94, pp. 164–175. ACM, 1994.

[RFC2693] Ellison, Frantz, Lampson, Rivest, Thomas, Ylonen. *SPKI Certificate Theory*. IETF, 1999.

[JLS:SE7] Gosling, Joy, Steele, Bracha, Buckley. *The Java Language Specification: Java SE 7 Edition*. Oracle, 2012.

[Kroyer08] Kent Krøyer. *Signalsystem for 24 milliarder: Skærm i førerkabine afløser lamper ved skinner*. Ing.dk, 2008.

[Chen76] Peter Pin-shan Chen. *The entity-relationship model: Toward a unified view of data*. ACM Transactions on Database Systems, 1; 9–36, 1976.

B Glossary

802.11n
: The latest 802.11 wireless LAN ("Wi-Fi") standard, finalized in 2009.

AP
: Access point (see pg. 2).

ASCII armor
: The OpenPGP standard for including binary cryptographic data (e.g. a digital signature) in a plain text file.

CFG
: Context-free grammar, the formal specification of a context-free language (see pg. 39).

CPL
: Class precedence list (see pg. 45).

CIDR notation
: A shorthand for specifying an IP address and an associated network mask (the latter specified as the number of leading one bits in the mask).

CRLF
: Carriage return and line feed, the ASCII control codes that indicates a Windows-style newline. Compare LF.

CSR
: Central system router (see pg. 2).

DSA
: Digital Signature Algorithm, a widely used digital signature algorithm.

GnuPG
: GNU Privacy Guard, an open-source OpenPGP implementation.

GSM-R
: Global System for Mobile Communications – Railway, a variant of the GSM mobile phone standard specifically designed for railway communications.

GUI
: Graphical user interface.

Kleene repetition
: Refers to a number of formal language operators, most importantly the Kleene star (denoted $*$ in regular expressions) indicating that a construct may appear zero or more times.

LF
: Line feed, the ASCII control code that indicates a Unix-style newline. Compare CRLF.

LWSP-char
: Linear white space character, an ASCII space or horizontal tab control character ([RFC822]).

NMS
: Network Management System (either the present NMS or NMS-NG).

NMS-NG
: Next Generation NMS, the new NMS described in this report.

OOP
: Object-oriented programming.

OpenPGP
: A standard for document signing and encryption (as well as key management), defined in [RFC4880] and related documents.

RFC
: A standards documents published by the Internet Engineering Task Force.

RSA
: A widely used algorithm for public-key encryption and digital signatures.

TU
: Train unit (see pg. 2).

VLAN
: Virtual LAN, the result of multiplexing multiple logically separate networks on a single physical network.

vtable
: Virtual method table, a mechanism for dynamic dispatch in OOP.

WUI
: Web user interface, specifically the NMS-NG web interface.