Australian Government
Department of Defence
Science and Technology

Cards Reference Manual

Daniel Finch[1], *Paul Berry*[1] *and Zane Van de Meulen-Graaf*[2]

[1] **National Security and ISR Division**
[2] **Simbiant Pty Ltd**
Defence Science and Technology Group

DST-Group–TN–1803

ABSTRACT

This document describes the use and design of the software tool Cards. Cards is a user interface for structured parameter inputs, and is easily customised to suit a wide variety of software and hardware systems.

RELEASE LIMITATION

Approved for Public Release

Produced by

National Security and ISR Division
PO Box 1500
Edinburgh, South Australia 5111, Australia

Telephone: 1300 333 362

August, 2018

APPROVED FOR PUBLIC RELEASE

Cards Reference Manual

Executive Summary

Cards is a user interface and visualisation tool for structured parameter data, such as inputs into scientific simulation software. Cards provides a generic interface that allows data to be defined in a tree structure of parameters, avoiding the need for complex custom input files or user interfaces. It is easily customised to suit a wide variety of software and hardware systems due to its flexible schema design and use of the Python language. It is hoped that by adopting Cards, significant effort and expense can be saved by reducing the need to develop custom GUIs.

Cards provides input checking, and stores a history of runs to aid to the user in simulation data management. Additionally, the concept of *common projects* is introduced to allow multiple software systems to be executed using a single set of parameter inputs. Plotting functions can be added to Cards easily, and use of the in-built integration with SIMDIS provides visualisation of inputs and outputs from the system.

This document describes the use of Cards and how it may be readily extended to support other systems.

This page is intentionally blank

Authors

Daniel Finch
National Security and ISR Division

Daniel Finch works in radar modelling and analysis in the National Security and ISR Division of DST. He joined DST in 2003 after completing degrees in Mathematics and Electrical Engineering at the University of Wollongong. In 2012 he completed a Master of Sciences at Adelaide University. His specialisation includes radar performance simulation and modelling.

Paul Berry
National Security and ISR Division

Paul works in the National Security and ISR Division of DST and has interests in estimation, optimisation and control applied to microwave radar engineering. He has a PhD in Theoretical Fluid Mechanics from UCL, University of London and previously worked in research laboratories in the UK's energy industry on problems of computational physics and power system optimisation and control.

Zane Van de Meulen-Graaf
Simbiant Pty Ltd

Zane Van de Meulen-Graaf received a Bachelor of Mathematical and Computer Sciences (Honours) in Applied Mathematics from the University of Adelaide in 2008. He has worked as a contractor at DST since 2010 in various software-related roles, using a variety of languages and tools, including C++, Python, Go, and Java. He is currently working on high-performance, real-time signal capture and analysis software in C++.

This page is intentionally blank

Contents

Figures

Tables

Glossary

Backend	Software written for Cards to translate parameters for a system
DST Group	Defence Science and Technology Group
GPARM	Radar simulation software created by DST
GUI / UI	(Graphical) User Interface
MATLAB	A numerical computing environment and programming language
Miranda	Radar analysis tool developed by DST
Model	A collection of parameters to represent a specific item or functionality
Parameter	A named item that is used to take on a value or create the structure of the inputs
Project	A collection of parameter values used to run a system plus any outputs from the run
Python	A programming language
Schema	Definition of the parameters for a model, including name, data type and restrictions
SIMDIS	3D visualisation tool developed by U.S. Naval Research Laboratory
System	The software or hardware process executed by Cards
System inputs	Inputs in the format accepted by the system, as opposed to the Cards format

This page is intentionally blank

1. Introduction to Cards

Cards is a generic graphical user interface (GUI) that can be customised to command a wide variety of non-interactive software or hardware systems. Such systems are widespread in scientific and research organisations such as the Defence Science and Technology (DST) Group, and often require a large number of inputs to operate. These systems subsequently require their own GUI to be crafted or are configured using complex text files. Cards attempts to solve this problem by providing a generic interface that allows data to be defined in a tree structure of parameters. Values for the parameters can be manipulated by the user within Cards, then effortlessly passed to the target system in the required data format.

Cards was conceived as a tool to assist in radar analysis and modelling with the aim of integrating analysis, simulation & visualisation tools into a single interactive environment; allowing the analyst to use multiple tools without having to re-enter or convert input data. The design of Cards was influenced by the Miranda [1] and GPARM [2] radar models developed by DST.

While the main focus of Cards is to provide a unified GUI, it provides a number of other useful functions with regards to simulation:

Input checking of the user-entered data for each parameter. This provides type-checking (i.e. enforced data types) and the option for limits to be placed on values. Units can be specified for numeric data types. This checking reduces the amount of input validation required at the system level and improves usability.

Run History that keeps metadata about runs that have been performed through Cards. This metadata is easily searchable, and provides additional utility by allowing input parameter files from multiple runs to be compared, leading to simple and fast checks regarding parameter variations between runs.

Common Projects that allow multiple systems to be controlled from a single set of inputs. This can be useful when attempting to compare the performance of two or more systems.

Plots can be produced both pre-run and post-run, depending on the system being used.

SIMDIS Visualisation of a given scenario. SIMDIS [3] enables two and three-dimensional interactive graphical display of data and can be used to help verify that the scenario geometry is correct.

This document describes version 16.06 of Cards, released in June 2016. Section 2 gives an overview of the basic usage of Cards, and further functionality is described in Section 3. Details of extending Cards to work with new systems, including creating schemas and adding units, is detailed in Section 4.

1.1. Getting Started

Cards is written in Python, and is hence cross-platform. PythonXY 2.7.6.0 was used in the development and testing of Cards, however any Python 2.7 distribution should be suitable. The list of packages required by Cards is given in Appendix A. More information about Python can be found at the Python website www.python.org/.

Once Python is installed and the Cards files[1] extracted to a suitable location, Cards can be launched by using the command

```
python -m cards
```

from the command line within the Cards root directory. This will open the main GUI, an example of which is shown in Figure 1.

The GUI has three key areas: i) the left *navigation pane* displays the parameter hierarchy as a tree, ii) a *parameter pane* on the right displays the child parameters of the branch currently selected in the navigation pane, and iii) a log pane at the bottom. The use of the GUI will be described in more detail in Section 2.

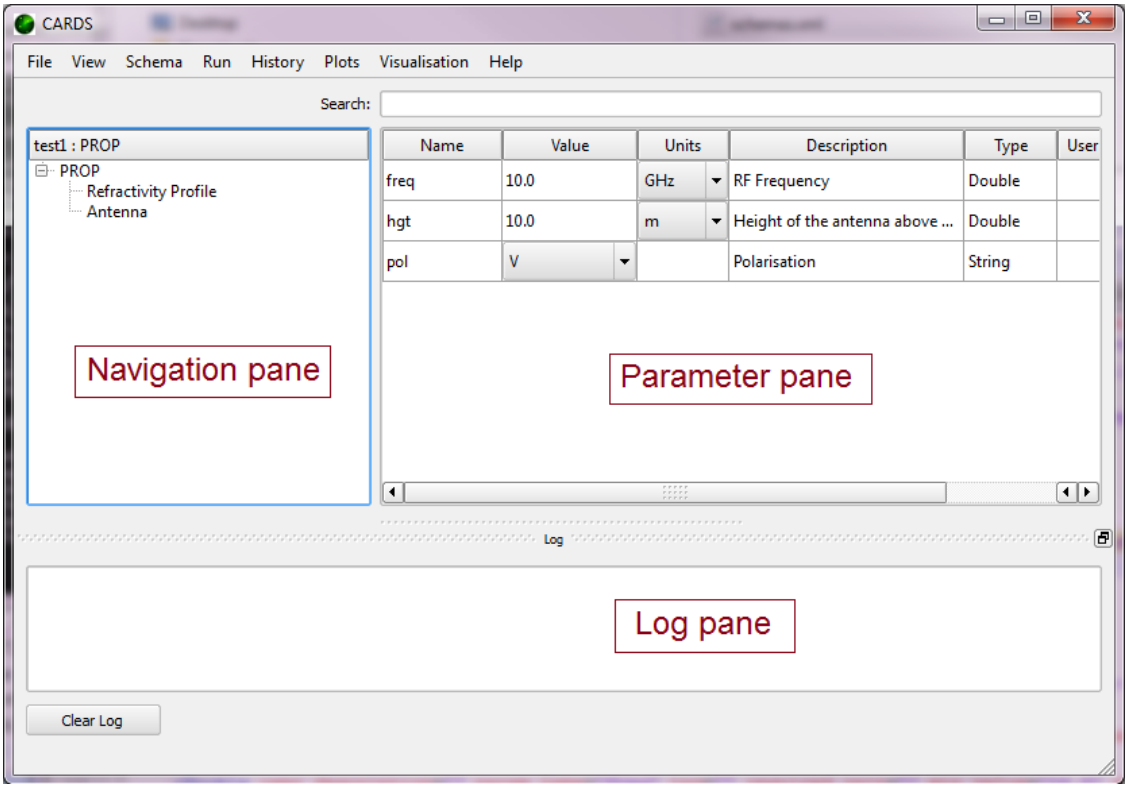

Figure 1: Cards GUI with project open for example system

[1] Cards is distributed as a zip file containing the Python source and other resource files. Requests for copies of Cards should be made to the author in the first instance.

1.2. Functional Overview

Cards performs a number of functions as outlined in Section 1, but its core purpose is data input and system execution.

Systems are the end software that consume the data defined in Cards, and may be executables, MATLAB functions or remote processes. Cards' responsibility is to:

1. present a set of parameters to the user for data entry via the GUI
2. transform the parameter inputs into the format required by the system
3. invoke the code to execute the system with these inputs
4. report any execution errors
5. log history on success.

In Cards this process is termed a *run*.

Parameters required by each system are defined in a series of *schema files*. These files are interpreted by Cards and used to populate the navigation and parameter panes of the GUI. The user can then enter values for each parameter, which are later stored as part of a *project*.

Projects are a collection of files describing the user inputs and any subsequent outputs from running the system. Projects can be saved, closed and reopened within Cards, allowing inputs to be edited and runs to be repeated. A project has a single target system[2], e.g. a Miranda project cannot be used to run GPARM.

To perform a run, the user's inputs must first be translated to the format[3] expected by the system. This task is performed by the *backend* class. Once this is done the *system inputs* are transferred to the system for processing, and any outputs are copied into the project. The relationship between these items is shown diagrammatically in Figure 2.

1.3. Parameter Data Structure

The parameter data structure is an important aspect of Cards. A hierarchical structure is used, with a tree branching from a *root* element to its *child* elements. The terms parent, child, and sibling are used to describe the relationship between elements. This is combined with polymorphism (sub-typing) to achieve a dynamic structure. This structure nomenclature is further described with references to the parameter set for the fictional PROP system parameters shown in Figure 3.

[2]Unless it has been created as a *Common* project, which will be described in Section 3.2.

[3]This may be a file format e.g. XML, CSV, MATLAB (.mat), or serialised into a memory buffer for transmission to the system.

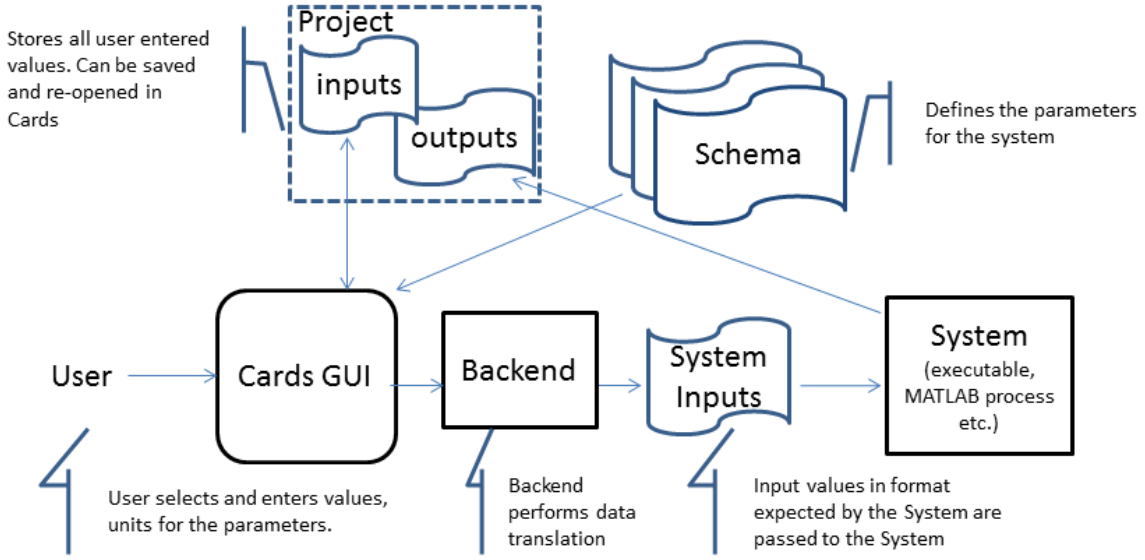

Figure 2: Relationship between elements of Cards, showing the schema, project, Cards UI, backend and the system

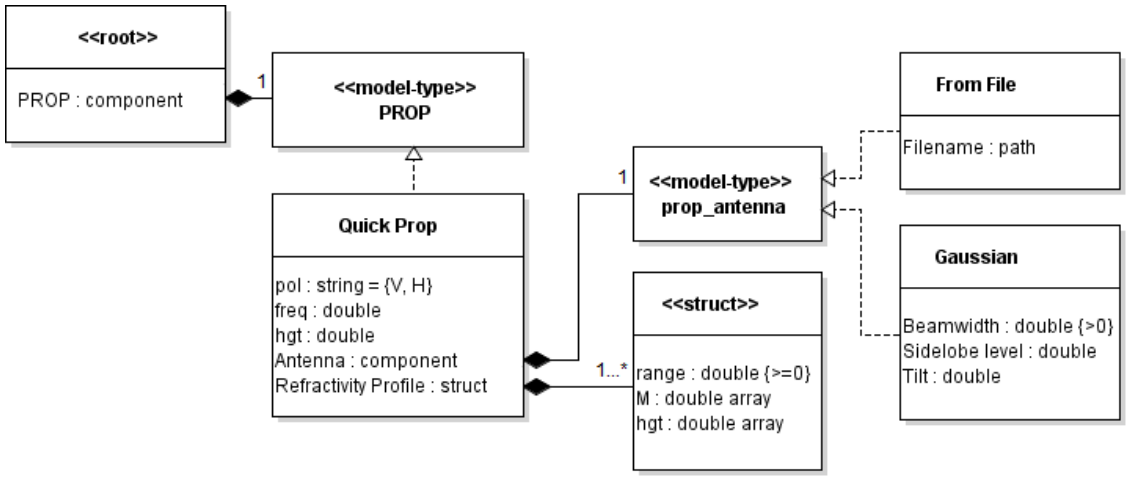

Figure 3: Example parameter structure for Cards based on the demonstration PROP system. The relationship is shown using the UML class style. Models are shown in the two-part boxes with names in bold text at the top, and parameters in the lower part of the box. Parameter names are followed by their data type and restrictions. Branch Parameters are shown using the composition (filled diamond) connection. The model type is shown in the single box, and the model options are shown using an implementation (dotted) connection.

Model A model is the basic building block used to form the tree structures within Cards. Models have a model name, a model type, and zero or more child parameters. In the example parameter structure *Quick Prop*, *Gaussian* and *From file* are all models.

Parameters A parameter is a child of a model. There are broadly two types of parameters: branch parameters, which may have their own children; and leaf parameters, which take a value but do not have any children. In the example figure, *pol* and *Beamwidth* are two of the leaf parameters, while *Antenna* and *Refractivity Profile* are branch parameters. Leaf parameters provide input validation and, for some data types, can have units. Branch parameters are shown in the navigation pane of the GUI, while the leaf parameters of the selected branch are shown in the parameter pane.

Data type All parameters have a name and a data type, the full list of data types is given in Appendix B. In the figure the data type of each parameter is shown after the parameter name.

Component This is a data type that acts as a slot (or interface) that other models connect to. The models that may be connected are defined via the model type. In the example structure the parameter *Antenna* is a component, with restriction to model type *prop_antenna*; and the user has the choice between two models for this model type. Components are a type of branch parameter but do not have their own pre-defined children, rather they take the children from the connected model.

Struct This is the second type of branch parameter, and provides an alternate way of creating a nested parameter structure. Structs have their own pre-defined child parameters which cannot be altered. In the example, *Refractivity Profile* is a struct with three child parameters.

Duplicates Branch parameters may be duplicated (cloned) if this functionality is permitted by the schema. Duplicating a parameter creates another instance of that parameter in the navigation pane. This can be used to define an additional set of inputs, for example multiple refractivity profiles at different ranges in the PROP system. To differentiate between duplicates, each instance is given a unique 'branch name' that can be set by the user.

Using the abstract parameter structure it becomes possible for the user to generate a variety of valid data sets via model selection and duplication. This is a key capability of Cards and allows the parameters to be described in a way that is easier for a user to understand. Using the options from Figure 3, a user may construct the example parameter sets shown in Figure 4. This demonstrates that the parameters presented to the user in Cards need not be a one-to-one mapping to the system inputs. The backend is capable of performing pre-processing on the inputs, and is not simply parameter mapping.

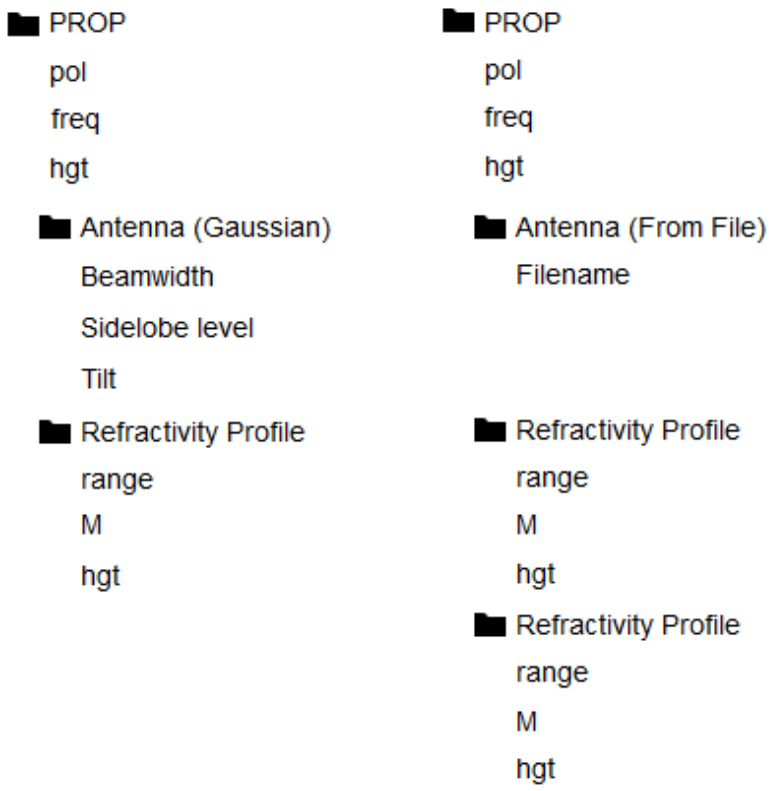

Figure 4: Two possible parameter structures based on the demonstration PROP system using a file/folder representation. The two structures have different models selected for the Antenna parameter, and the second has a duplicate of the Refractivity Profile.

2. Creating and Running Projects

The steps involved in using Cards are: creating a project, setting parameter values, and running the project. The following sections provide more information on each of these steps.

2.1. Creating a New Project

Projects are the mechanism by which Cards groups inputs and their outputs from runs. A project is targeted at a single system[4], which cannot be changed once the project is created.

Projects are created and loaded from the main Cards window via the *File* menu. Creating a new project will launch a dialog box requesting a project name and system selection. The project name is a label to help identify the project. It can be changed later and need not be unique (although this is a good idea), and does not relate to file or directory names when the project is saved. The project name and system will be displayed at the top of the navigation pane once the project has been created.

[4]Unless it is a common project.

Projects consist of multiple files and are hence saved as a directory. When saving a project it is a good idea to create a new directory, otherwise existing project files may be overwritten without warning.

2.2. Setting and Modifying Parameters

Once a project has been created, a root parameter with the same name as the target system will be added to the navigation pane. To select a model, right click on the parameter and choose an item from the *Select Model* list. When this is done the navigation and parameter panes will be updated with the child parameters of that model.

The parameter pane is organised as a table with six columns and a row for each leaf parameter. A description of each column is given in Table 1. Parameters are always shown in alphabetical order[5], and receive an initial default value defined by the schema.

Table 1: Columns of the parameter table

Column Name	Editable	Description
Name	✗	The name of the parameter
Value	✓	User entered data
Units	✓[a]	Drop-down box of options
Description	✗	A short summary of what this parameter does
Type	✗	The data type e.g. 'double', 'string'
User Description	✓	This is an optional free-text area. The user may add any comments here

[a] only if type is a numeric quantity and unit type is specified in the schema

When modifying parameters with units it is advisable to set the unit to the desired quantity first. Cards will automatically convert the existing data value to the new unit selected in the drop-down box. To modify a value, double-click on the cell containing that value. These inputs are type checked, so Cards will reject an input such as "hi" for a Double type, and revert to the previous value.

Hovering over a branch parameter will reveal its data type, branch name and other relevant information. If duplication of a branch parameter is permitted a *Duplicate* menu item will become available in its context menu. Likewise it is possible to remove a duplicate by selecting *Remove Duplicate*. The only limit placed on the number of instances of a duplicable parameter is a minimum of one.

To search for a specific parameter name, use the *Search* box above the parameter pane. Any matches will be highlighted firstly in the navigation pane, signifying that some parameter name under that branch matches, and then in the parameter pane when one of the matching branches is selected. Note that branch parameter names are not searched.

[5] This is regardless of the order that parameters appear in the schema.

2.3. Paths

Before Cards can be used to run a project it must know where the executable for the system is located. This can be set by via the *File* → *Paths* menu, with the UI showing two tabs: *Executable Paths* and *Schema Paths*. Executable paths contains a user-settable path for each system, as well as a *Diff Tool Directory* path (see Section 3.1.2). Schema paths are configured when adding systems or expanding existing systems, and are described in Section 4. Path information is stored independently of projects, hence should only need to be configured once.

2.4. Running a Project

Prior to executing a project it must first be saved, it is not possible to run an unsaved project. Note that Cards will automatically re-save the project each time the user initiates a run. The *Run* menu within the main Cards UI is used to initiate a run. While the project is running, the user may continue to use the GUI and modify parameters in preparation for the next run without affecting the current run. However the *Run* menu option will be disabled until the current run has completed.

Whilst the system is executing a small window will remain open showing a stylised radar display. Once the system has finished executing, Cards will close this window and information about the output path will be printed in the log pane[6]. If the run failed, information about the cause will be shown in the log pane.

Cards allows the user to terminate a run before it has completed. This is done via the *Run* → *Stop* menu and will result in a "run aborted by user" message in the log. For systems using an executable (e.g. .exe file), this should not cause any issues; however for MATLAB based systems this should be used with some caution as the MATLAB engine may not be given a chance to properly shut down.

3. Additional Features

3.1. Run History

Being able to reproduce results is incredibly important when simulating. Without such capability, it is impossible to perform any kind of validation of the model, greatly reducing the confidence that it is correct. Across many runs of a system, it can become difficult to remember which parameters were used for a specific run, and where those files are now located. Provenance of results is very important, hence it is essential to have a good data management plan. To assist with this, Cards integrates *run history*. This is basic metadata that is captured about every successful run on a given computer. This is performed automatically and is transparent to the user.

[6]The log pane should be visible at the bottom of the Cards window. It may however be hidden or revealed via the *View* menu

3.1.1. Searching

The run history may be searched on the following fields:

Date ranges Filter runs by the date range on which they were executed.

System(s) Which system(s) were used to perform the run. Note that selecting no system is equivalent to selecting all systems.

Project Name The name of the project that was used to run the system.

Parameter search This will search the run history for a matching parameter name with value equal to or within a range specified by the user. Care should be taken when using this search as units cannot be specified for numeric values, and all strings are case sensitive.

Any combination of these fields can be used to define the search, unused fields should be left empty.

3.1.2. Results

Runs matching the sarch criteria will be listed in a results window with basic information about each run. The user can interact with these results in a number of ways via the context menu:

Remove Run This will remove the run from the run history database. This will not remove any files associated with the run.

Save Parameter File Will save the inputs that were used to generate the run to a user selected location in the Cards XML format. This can be useful for quickly viewing the inputs rather than recreating the project.

Open Output Location This will open the computer's file manager application to the project directory for that run, if the folder still exists. Note that subsequent runs of the project may have replaced the output files.

Recreate a project This will create a new project that can be opened in Cards so the run can be repeated.

Diff Parameter Files If exactly two runs are selected, a comparison on the parameters can be performed. An external program is used for comparisons; for Windows WinMerge[7] and TortoiseMerge[8] are supported. The desired tool must be installed on the user's system and the *Diff Tool Directory* path, discussed in Section 2.3, must be configured prior to performing a comparison.

[7] http://winmerge.org
[8] https://tortoisesvn.net/TortoiseMerge.html

3.2. Common Projects

Common projects are designed to simplify the process of executing multiple systems using a single set of parameters. Whereas a normal project can only target a single system, common projects can target many systems. To achieve this, the common project is composed of: a base project for each system, a set of common parameters, and a mapping of these common parameters to those in each base project.

Common projects can also be used to create an alternate input parameter set for a single system. This can be a useful way of limiting the number of parameters that a user may edit, without the need to modify the backend for the system.

3.2.1. Overview

To understand what a common project does, consider the following simple example where we wish to run two similar models (System A and our fictional PROP system), to compare their performance. As shown in Figure 5 we want to allow the user to specify the height and polarisation parameters only. To do this, a project for each system must first be created such that the desired scenario is represented. These become the *base projects*.

After the base projects are created the desired *common* parameters must be defined and mapped back to parameters in the base projects. It is these common parameters that the user will enter data for; in this example they have been called *Height* and *Polarisation*. The mappings of these parameters are shown in the figure.

Once the parameters and mappings have been defined, a *common project* must be generated. This project can then be opened and edited in Cards much like any other project. When the user runs the project, the entered values for the common parameters are merged back into the base projects, creating a complete parameter set for each system. These merged projects are then run (in parallel if possible) and the outputs saved into the common project. Plotting options for both systems will be available under the *Plot* menu, however there is currently no mechanism to produce a combined plot or perform other processing of the different system results.

3.2.2. Limitations of Common Parameters

There are a small number of limitations associated with common project parameters. Firstly, due to the parameter mapping process, typically only one parameter in the base project can be mapped to each common parameter[9]. That is, there is an assumption of independence between the parameter selected from the base project, and other parameters within that base project. Secondly, it is not possible to modify the common value input before it is merged into the base process. This may be an issue for say the string input parameter *Polarisation* from the previous example. If system A expects 'Vertical' for vertical and PROP expects 'V' then this cannot be done in Cards. Finally, the common project cannot have any branch parameters, as the mapping required for this is too difficult.

[9]There is an exception if the parameters in the base project have identical names

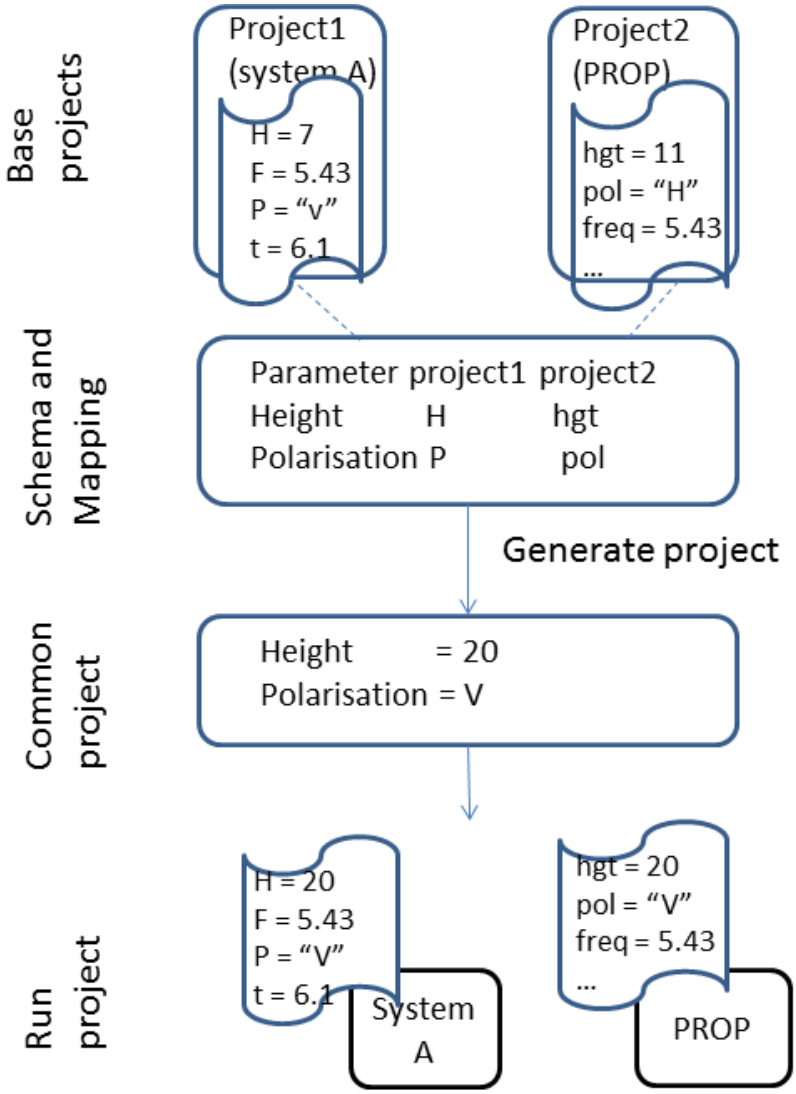

Figure 5: Common project example

3.2.3. Creating a Common Project

This section will step through the process of using Cards to create and run a common project. As identified earlier, the common project requires a base project for each system to be executed. These should be created in the normal way using Cards and saved to suitable locations.

The Common Schema Editor is used to define the parameters and their mappings to the base projects. This may be launched via the *Schema* menu in the main Cards UI. The *Add project* button is used to select the base projects. Each base project will be represented as a column in the editor, with the project name as a header. The desired common parameter names are then entered in the first column and base project mappings in the remaining cells. Using our earlier example projects, the UI will resemble Figure 6.

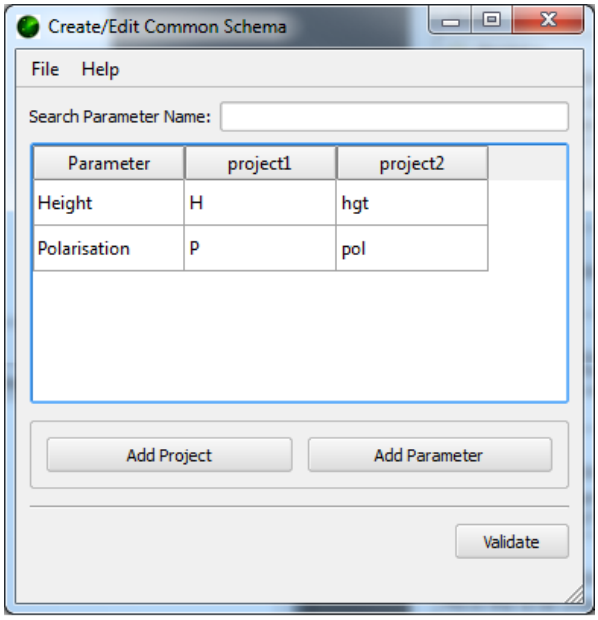

Figure 6: Common project parameter mapping UI

In the bottom right-hand corner is a *Validate* button that is used to perform a number of checks on the entered mappings. The validation firstly ensures all parameter names exist in the base project; from the previous example, project1 would be checked to ensure that the parameters named *H* and *P* exist. If any of these parameters are not found, validation will fail and the cell will be highlighted in red. Note that parameter names are case sensitive.

The second stage of the validation is a check for ambiguous parameters. It is possible for the base project to contain two or more parameters with the same name, either through duplication of a branch, or across unrelated branches. In this case Cards needs assistance from the user to correctly resolve the parameter, and will highlight the parameter in yellow. Right-clicking over any highlighted parameter will bring up a list of 'fully resolved names' that may be selected to disambiguate the mapping[10] Ambiguous parameters will only trigger a warning; it will not stop the schema from validating. If the parameter name is left ambiguous, Cards will map to all instances, and each of these parameters will be updated with the given value.

The final check performed during validation is to ensure that the data types and unit type (if defined) of the parameters from the corresponding base projects are identical. From the previous example, this will check that the data type and unit type of parameter 'H' in project1 is the same as 'hgt' in project2.

Once the common schema has been validated, the user can then generate a new project. To do this, use the menu item *File* → *Generate Common Project*. Following this, the schema UI may be closed and the new project loaded in the main UI as per normal.

[10]The format of a fully resolved name will be of the form <parameter>@<branch name>.<parameter>. Branch names can be set via the context menu in the navigation pane of the UI.

3.3. Plotting

There are two types of plots that can be produced from Cards: *pre-process* and *post-process*. The pre-process plots are those that can be produced using only the input data for the system. These plots can be created at any time, and may be useful as a check of the input values. The post-process plots use the outputs of the system, and can only be produced once the project has been run.

The plot production is defined as part of the backend class for each system, hence the plots available will vary between systems. The process of adding new plots for a system is described in Section 4.4 and uses the matplotlib [4] Python library. Figure 7 shows an example of the quality of plots possible in Cards.

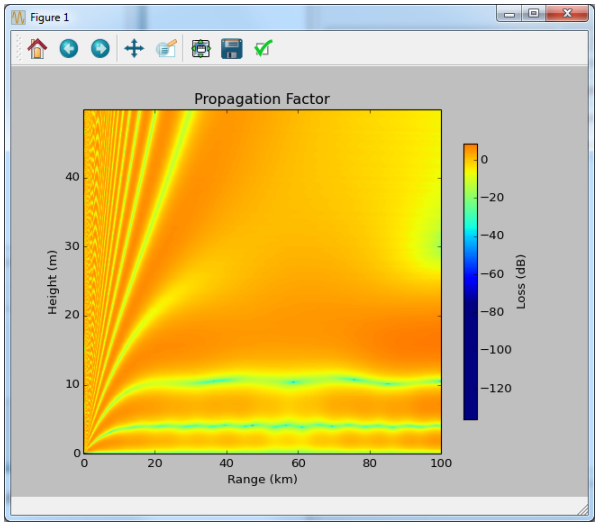

Figure 7: Example plot produced by the Cards using the matplotlib Python library

3.4. SIMDIS Visualisation

SIMDIS [3] is a two and three-dimensional interactive visualisation tool developed by the U.S. Naval Research Laboratory. A plugin for SIMDIS is included with Cards that enables data from Cards to be sent to SIMDIS, allowing users to update parameters and see the changes reflected immediately in SIMDIS. This can be used to verify the scenario geometry is correct. An example screen shot of SIMDIS is shown in Figure 8.

To enable a platform to be displayed in SIMDIS, the Cards schema must be configured with appropriate tags which provide a mapping of the Cards parameters to those expected by SIMDIS. This will typically be performed by the schema developer, and is not described further here.

Prior to using SIMDIS, users must ensure that SIMDIS 9 is installed and registered on their computer. The SIMDIS directory must be added to the path environment variable. Once a suitable Cards project has been created and populated with values, SIMDIS can be launched using the *Visualisation* menu. Within SIMDIS, the Cards plug-in must be installed using the

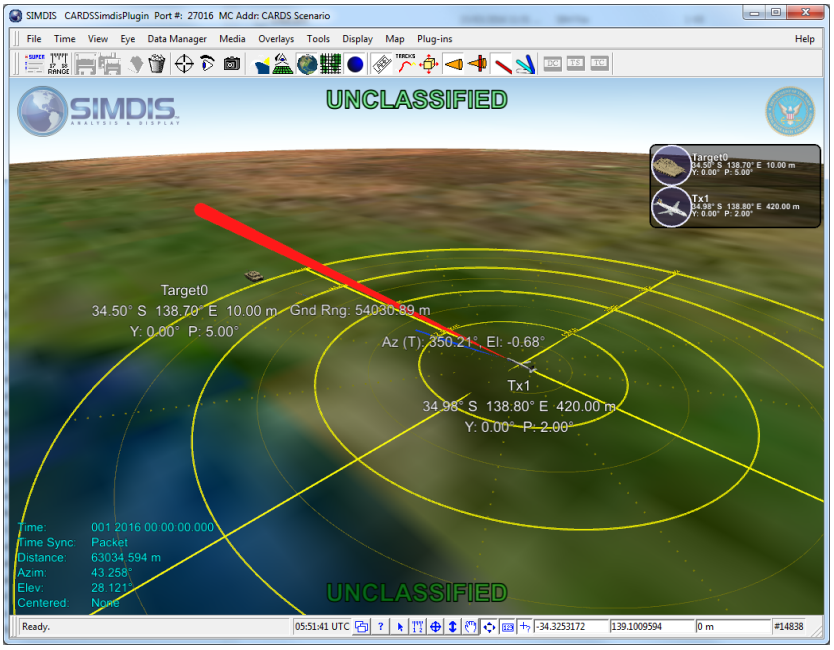

Figure 8: Example SIMDIS display showing a single airborne transmitter and ground based target. SIMDIS enables the addition of range and angle information to the display as shown. Low resolution map shown, but higher resolution may be used.

plug-in manager[11] . Once the plug-in is successfully installed, the SIMDIS eye will reposition itself to be above Adelaide, Australia. With SIMDIS running, users can modify the tagged parameters in Cards and view the changes in SIMDIS.

4. Extending Cards

4.1. Adding a System

This section is designed to highlight the key steps necessary to integrate a new system into the Cards framework. Cards was designed with flexibility in mind, and this process is intended to be as painless as possible.

The broad steps needed to integrate a new system are:

1. Create a backend class for the new system derived from the *System* interface
2. Register the new class with the system manager
3. Set a system path
4. Create a set of schemas.

[11] This may need to be done every time SIMDIS is launched as it appears to lose this setting

The remainder of this section describes the first three steps using our exemplar PROP system. The files to be created or modified will all typically sit under the *backends* directory, except for *config.py* which is directly under *cards* itself. The schema generation is described in Section 4.2.

4.1.1. Creating the System Class

The first step in integrating a new system into Cards is to create a system backend class. There are no restrictions on the name of the class; however this name will be used by Cards as the system name in most cases. Using the PROP example, the current convention would be to create a module named *prop.py* in the *backends* directory[12], and within this module create a class called `PROP`.

Cards defines an abstract class in *backends/system.py* called `System`. In order for the system being integrating into Cards to work it must derive from this class. The `System` interface defines an abstract property and two abstract methods, in addition to this, the new class must have an `__init__` method that takes `project_directory` as an optional argument. The role of these properties and methods are discussed in the following paragraphs. A skeleton of the PROP class is provided in Appendix C.

`system_name`

This method should return a string containing the name of the system. In the majority of the Cards code the system name is derived from the name of the class rather than this method. To avoid confusion it is best practice to ensure the string returned by this method matches the class name.

`export`

The export method is responsible for transforming the Cards parameter structure to the system input representation. This format is entirely dictated by the system and assumed to be known to the developer. This method is called following some basic sanity checks on the data when the user initiates a run or chooses to export via the *File* menu. For systems that accept inputs via file, the export method will save the file to the project directory. The two arguments are:

`root_parameter` is the root parameter as shown in the GUI's navigation pane. Whilst only this parameter is passed, it contains all the parameter information through its children.

`project_directory` is the full path to a project directory to which the system input file(s) will be exported.

`run`

The `run` method is called by Cards to execute the system once the data has been exported. The parameters are:

`system_directory` is the directory defined by the user in the path setup. This is the directory

[12] module names should always be lower case

that contains the executable code for the system.

`project_directory` is the full project path. It is recommended that any input files are saved in this directory during `export`.

`enable_run_signal` is a signal that will re-enable the *Run* menu in the main Cards UI. To allow multiple parallel runs of the system, simply emit the signal as soon as the run function is entered, otherwise it should be emitted once execution has completed.

`log_run_signal` is another signal. It is used to log a successful run into the run history.

`stop_signal` will be emitted on if the user elects to stop a run before it is complete.

`finish_func` is an optional function that will be run on completion. This is typically unused except for the Common system, and can be safely ignored.

It is advisable to have `run` spawn off a separate thread that performs the run, and then returns. This is to stop the UI from blocking, preventing the user from using Cards until the run is complete.

There are two helper classes available, `RunExecutable` and `RunMatlab` that may be used to perform the majority of the repetitive setup and teardown. Information on the use of these classes is contained within the files *executable.py* and *matlab.py* in the *backends* directory.

4.1.2. Registering the Class

Registering the class is a simple process to let Cards know about the new system. This is achieved by importing `SysManager` from *backends/system_manager.py* and inserting the statement `@SysManager.register` directly above the new class defined in Section 4.1.1. Finally, in *backends/__init__.py*, add the module that the new class lives under to the import list.

4.1.3. Setting a System Path

Cards assumes that all systems require a path setting that informs the backend where the system code and/or executable can be found. This is the `system_directory` argument passed to the `run()` method. This user defined setting is stored within the *path_preferences.cfg* file and accessed by the *PathManager* class.

Adding an entry to the .cfg file for the new system involves creating a new <*name*> = <*value*> line below the `[ExePaths]` section. If the system does not require a path the <*value*> may be left empty.

To enable Cards to associate an entry in the .cfg file with a system, the *config.py* file needs to be updated. This requires an addition to the `SYSTEM_TO_DIRECTORY` map located under the *Directory Configuration* section. The new entry should have the system name as the key and <*name*> used in the .cfg file as the value.

The modifications to the .cfg and *config.py* files for the PROP system are shown below

```
[ExePaths]
miranda_dir = C:\miranda
prop_directory =
```

```
SYSTEM_TO_DIRECTORY = {'Miranda':'miranda_dir', 'Common': '',
                       'PROP': 'prop_directory'}
```

To enable a user to modify the path preference via the UI requires modifications to *ui/paths.py* and the associated view using *Qt Designer* software. These modifications are beyond the scope of this document.

4.2. Schema Generation

4.2.1. Structure

Cards is designed to be a front-end for various systems with disparate parameter sets. As such, a method of defining the set of input parameters for each system in a standardised format is required. This is the role of the schema. Further to this, schemas enable a number of other useful features such as input validation, unit restrictions, and parameter descriptions.

New schemas are required to be created when a system is integrated into Cards for the first time. It may also be possible to extend the functionality, or improve the usability of an existing system by creating new schemas.

This section covers the basics of defining a schema, but not how Cards should translate this information into the system input. This translation is performed in the Cards backend class for the system (in the `export` method), and is beyond the scope of this document. Examples of how this may be achieved can be found in the existing backend classes.

A schema is a description of zero-or-more parameters, and a separate schema must be defined for each model; hence a system may require many schemas to be defined. Each parameter within a schema must have a unique name, a data type and a description. Depending on the data type of the parameter, a number of other properties may be specified in the schema. The list of available data types is provided in Appendix B.

Schemas are saved as XML files and are typically kept in a subdirectory of *cards/schemas/*. They may however be placed in any directory so long as that directory is added to the schema path, which may be edited using the paths UI. Cards will recursively scan subdirectories of all items on the path for schema files; however it will not allow you to add paths that do not contain any schema files.

4.2.2. Creating a Schema

A Schema Editor UI is provided in Cards to simplify the process of creating and editing schemas. An example of this UI is shown in Figure 9 and can be accessed by selecting *Schema → Edit Schema* from the main UI menu.

The region at the top of the UI is the model identification area, and below that is the parameter definition area. The model identification consists of:

Model Type specifies the model type of this model. This should match the 'Model Type' property of any parent parameters. To create a root model, the value should be set to the system name.

Model Name specifies the name of the model, and will be shown under the *select model* context menu in the main UI when a user is editing a project.

The parameter definition area is composed of the *parameter table* and a set of controls for adding, removing and modifying parameters. The parameter table shows all parameters currently added to the schema. There are five columns in the table: *Name, Default Value, Units, Description,* and *Type*. Any of these properties may be edited within the table, however it may be easier to use the *Modify* tab located below the table[13]. A *Search Parameter Name* control above the table allows a basic search over all the names of parameters that have been added to the current schema. Anything that matches will be highlighted.

The *Insert Item* drop-down located below the parameter table is where the data type of a new parameter is selected. Once the data type is selected the name and other properties of the new parameter can be entered using the *Insert* tab. The new parameter is added to the parameter table by clicking the *Add Row* button. The *Add Child* button is used if the new parameter is to be added as a child of a struct parameter that has been selected in the parameter table.

4.3. Managing Units

Cards was created with a mechanism to allow conversion between different units. Support for common physical quantities and units is built into Cards, with a full list available in Appendix B.4. However, in the event that a desired unit (or quantity) is not provided then these units can be easily added.[14]

A full example of how this can be achieved is shown in the following sections. All the unit additions are contained in the file *cards/conversion_additions.py*, with the modifications to sit in the `add_conversions` method.

[13] The data type may be modified in the parameter table via the context menu.

[14] This capability may also be useful if the default range of units does not naturally align with a parameter, e.g. having km as a unit for the thickness of paper.

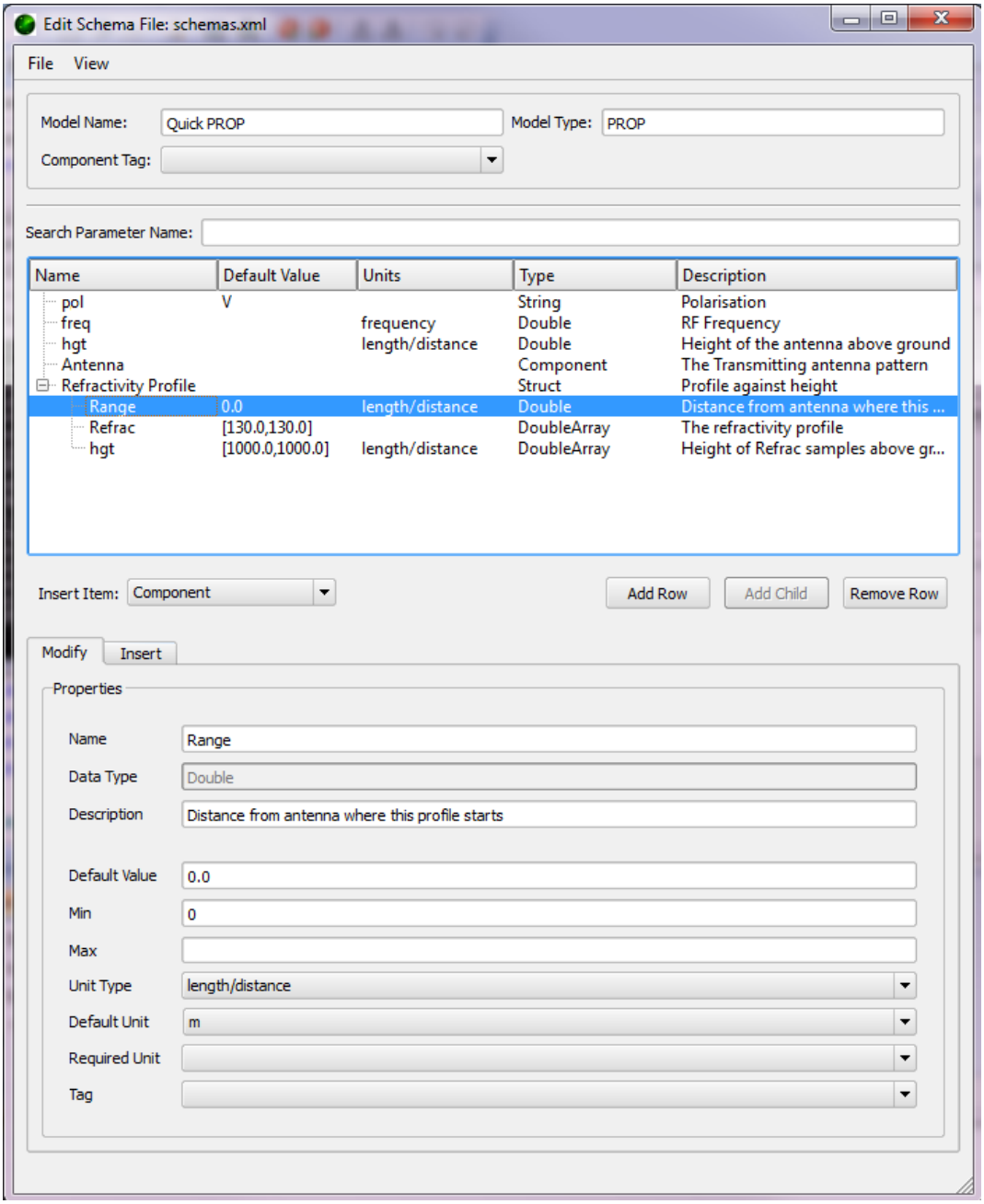

Figure 9: Cards schema editor showing the 'Quick PROP' model for the example PROP system.

4.3.1. Adding a New Quantity

In Cards a physical quantity is called a *unit type* and specifies a set of units that can be converted between. For example *angle* is a unit type, with units of degrees, radians, milliradians, etc. The unit conversion is performed as a two step process in Cards, with the *canonical unit* acting as the intermediary. The selection of the canonical unit is something of an arbitrary choice; however it will affect the function transforms to be defined.

When adding a new unit type, the name of the type, a set of units and the canonical unit must all be defined. This is achieved using the `add_unit_set` function, which takes these three items as arguments. Using *density* as an example, with units of kg/m^3 and g/cm^3, plus g/m^3 as the canonical unit, this looks like

```
density = add_unit_set('density', 'g/m3', ('kg/m3', 'g/cm3'))
```

Once the list of units has been set we proceed by defining the conversion between them. This is where the canonical representation comes into play; we only need to define a conversion function and its inverse between the canonical unit to each of the units in the set. These conversions are registered using two methods on the quantity object: `add_cannonical_to_other` and `add_other_to_cannonical`. Each of these functions take two parameters: the name of the unit to convert from or to, and the function to do the conversion.

Consider the conversion from g/m3 to kg/m3. This is from the canonical unit to another unit, and the formula for converting is to divide the value by 1000. This can be defined in code as[15]

```
density.add_cannonical_to_other('kg/m3', lambda x: x / 1e3)
```

To finalise the definitions a call to the `finish` function is added. Internally, this performs a number of checks to make sure everything is as it should be. If everything has gone well, when Cards is restarted a message should appear in the log window: `INFO: User unit conversions successfully loaded.` In the case that an error is detected the message `ERROR: Some user defined unit conversion(s) failed!` will be logged along with some diagnostic messages. A description of these errors is contained in Section B.5.

The complete code for adding the example *density* unit type is shown below.

```
def add_conversions():
    density = add_unit_set('density', 'g/m3', ('kg/m3', 'g/cm3'))
    density.add_cannonical_to_other('kg/m3', lambda x: x / 1e3)
    density.add_cannonical_to_other('g/cm3', lambda x: x / 1e6)
    density.add_other_to_cannonical('kg/m3', lambda x: x * 1e3)
    density.add_other_to_cannonical('g/cm3', lambda x: x * 1e6)
    density.finish()
    return True
```

[15] The Python lambda form has been used to construct an anonymous function.

4.3.2. Extending Existing Quantities

This is the process of adding a unit to a pre-existing quantity (unit type) within Cards. The steps for adding this unit are similar to those for defining a new quantity, with the code inserted into the same method and an identical interface used. The one difference is the use of the method `extend_unit_set` to define the new unit. This function takes the two inputs: the unit type, and a list of the new units.

Consider the example where we wish to include *turn* as an angle unit (where 1 turn = 360 degrees). As before, we have two conversion functions to define: one to convert from the canonical unit for angles (degrees) to turns, and one to convert from turns to degrees. The methods to register these conversion is the same as for adding new units. The following code shows the complete definition, including a call to `finish`.

```
angles = extend_unit_set('angle', ['turn'])
angles.add_cannonical_to_other('turn', lambda x: x / 360.0)
angles.add_other_to_cannonical('turn', lambda x: x * 360.0)
angles.finish()
```

4.4. Plot Addition

This section is designed to provide a brief description of the process for incorporating new plots into Cards. There are a small number of plots already included in Cards; however the goal when designing the current plotting code was to provide an interface and some simple helper classes to make incorporation of new plots as easy as possible.

The process for adding a new plot to Cards involves creating a plot class, then registering it for the applicable systems. This is described in more detail below, along with a suggested plotting pipeline for improving consistency and re-usability of the plot code. The code discussed in this section lives under the *plot* module, which in turn sits under the main *cards* directory.

4.4.1. The Plotting Interface

Cards defines a pair of empty classes in *plot_types.py*, named *PreProcess* and *PostProcess*. A class that is to be used for plotting must inherit from one of these two, as they tell Cards whether a plot is available before and/or after a successful run.

For pre-process plots, the plotting class will be initialised (`__init__`) with two arguments:

`system_name` is the string name of the system requesting the plot.

`root_parameter` is the root parameter as shown in the GUI's navigation pane. Whilst only this parameter is passed, it contains all the parameter information through its children.

Once the class has been initialised, it will have `plot()` called on it. This method takes no arguments, hence the data extraction should take place during initialisation.

The post process interface is simpler, with the initialisation receiving only the filename of the output file containing the results generated from a run. The `plot()` function will again be called immediately after the class has been initialised.

An example plot class is shown in Appendix C for our exemplar PROP system.

4.4.2. Registering Plots

In order for Cards to know what plots are available, they need to be registered and identify the systems supported. The underlying mechanism for this is very similar to the way that systems are registered, as discussed in Section 4.1. To do this, `@PlotWrapper` is placed above the plotting class with the system class (not name) within parenthesis following. If the plot can be used with multiple systems, these may be listed in the parenthesis as a comma separated list.

Finally the Python module containing the plot class must be imported into the module-level *__init__.py* that sits directly under the plot directory. This ensures that Cards will register the plot class as soon as the plotting module is loaded.

Once the plot class is correctly registered and Cards restarted, the new plot will become available under the *Plot* menu, identified by the name of the plot class.

4.4.3. The Plotting Pipeline

When it comes to plotting, there is a relatively general pipeline that can be followed for every plot, this is shown in Figure 10. Cards does not enforce the use of this structure, only the plot interface must be adhered to, but Cards does provide some additional help if this pipeline is followed.

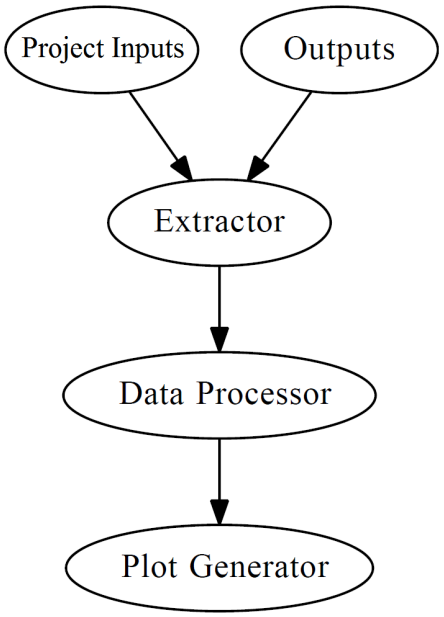

Figure 10: Suggested plotting pipeline

The first stage of the pipeline starts with either a project's input file, or an output file. This contains some super-set of the data required for the plot.

The second stage of the pipeline is an *Extractor*. For a given plot, there are usually only a few variables of interest; hence the idea of an extractor is to take the input and extract from this data only those variables. This decouples the rest of the pipeline from the input data format, meaning that multiple systems potentially only need different extractors; with the data processing and plotting code being reused across multiple systems.

Cards provides a base type *CommonExtractor* that can be used to aid in extraction of data for pre-process plots from the internal Component/Leaf input representation. For post-process plots, since all systems use different output file formats, most of this code will be specific to each system and it is impossible to create a more generic solution.

The third stage in the pipeline is completely plot-specific. It should take the input data in the format output by the extractor and processes it into a format suitable for plotting. This is the "number-crunching" step, and care should be taken to prevent this blocking the UI (making it non-responsive to the user) if this step takes significant time.

The final stage of the pipeline is the one that actually produces a plot on screen for the user.

5. Architecture and Development Roadmap

5.1. Technology Selection

Cards was designed with the following requirements

- use freely available Open-source tools
- high-level language for ease of development & maintenance by the user community
- provide good data visualisation options
- be interactive for ease of use.

Python was chosen as the language for Cards because it offers support for scientific libraries: numpy and scipy, is freely available, and the widely used PyQt enables GUIs to be developed easily.

Cards was designed to be deployed onto the end-user's computer; hence the selection of a language that transfers easy to a wide range of operating systems was important. Using Python provides this capability, and cards requires only a few non-standard packages to be installed as described in Appendix A.

5.2. Project Data Definition and Management

Schema files define the parameters for a model and are saved in an XML file format. When a project is saved, the schema definition for all models employed by the project are saved as part of the input file. This was done for simplicity, however it creates redundant data within the input file. That is, rather than referencing the schema file for the parameter definition, this definition is copied into the input file. Subsequently, when an existing project is opened, the original schema files are not accessed, rather the parameter definitions are read from the input file. A key outcome of this design is that a project's input file closely resembles the schema file. A brief description of the XML definition format is included in Appendix B, and a example schema file for a PROP model is given in Appendix C.

When a project is run, the output from the system is copied into the project directory. Unfortunately in the current design this will result in any outputs from previous runs being over written. The current design also introduces the possibility that the Cards input stored within a project may not be that which produced the stored output. It is suggested that the user employs their own data management or revision system to manage this process. This is a shortfall in the current design that will be rectified in future releases.

Users should be aware that the run history is stored within a single database, and not with the project. This spread of data may represent a risk when used in some environments. Alternative mechanisms are being investigated for future releases.

5.3. Future Development

Cards is still a young product, and as such it is expected to evolve quickly as the number of users increases and new systems are added. The list below outlines some of the planned enhancements to Cards in no particular priority order:

- Integration with simulation management tools. The history system in Cards has a number of limitations that may be fixed by integrating data/simulation management tools such as Sumatra.[16]

- Duplicated Parameters with order dependence. This is expected to allow Cards to support systems such as signal processing chains which require multiple components to be defined in a logical order.

- Batch runs and scheduling runs. The intent is to enable multiple runs to be defined prior to execution. The scheduling of the execution may then be handled by a workload manager such as slurm.[17]

- Improvements to the UI, including a framework for providing system help documentation to the user.

[16] https://pythonhosted.org/Sumatra
[17] https://slurm.schedmd.com/

These enhancements are expected to improve the usability and make Cards compatible with a greater range of systems.

Acknowledgements

Thanks to project AIR 7000 for providing funding and support for this project. Thanks to Matthew Wills and Jamie Grafton who performed the initial scoping and GUI design as part of their Summer Vacation Student project. Thanks also to Mr Gavin Currie and Dr Luke Rosenberg for reviewing this report.

6. References

[1] D. Finch and D. Gustainis. The Miranda modelling framework. In publication.

[2] P. E. Berry, G. Currie, and D. Yau. A generic phased array radar model for detailed radar performance assessment. 2011. 19th International Congress on Modelling and Simulation, Perth, Australia, 12 - 16 December 2011.

[3] Naval Research Laboratory. SIMDIS visual analysis & display. SIMDIS Web site https://simdis.nrl.navy.mil/ Retrieved 14 January 2016.

[4] J. D. Hunter. Matplotlib: A 2D graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

Appendix A. Required Packages

The following Python packages are required by Cards. Only those packages that are not typically provided with a *standard* Python distribution are listed. Note also that these packages may have their own required packages, which are not listed here.

- ConfigParser
- google
- matplotlib
- mlabwrap
- mpl_toolkits
- numpy
- pylab
- PyQt4
- pythoncom
- scipy
- sip

As a convenience, for computer systems where access to the Python Package Index (PyPI) is not available, the packages not distributed with PythonXY 2.7.6 are provided under the *cards\thirdparty* directory as *required_3rdparty.7z*. Simply extract this into *$INSTALLDIR\Lib\site-packages* where $INSTALLDIR is the Python installation directory, by default C:\Python27.

This page is intentionally blank

Appendix B. Data-types, Schema and Units

B.1. Summary of types

Cards contains the nine data types listed below. The first six of these are *basic* types, and the final three are *composite* types. A brief description of the validation rules is also given for each type.

Boolean a basic type that takes on one of two values, either True or False. Any capitalisation of the input is ignored, further it will accept any integer, with 0 being False and any any other value being True (much like most C-based languages).

Complex a type that signifies a complex number, of the form $(x + yi)$ (or $(x + yj)$). Parenthesis are optional; however the order of the y and i (or j) is not e.g. $3 + i4$ will be rejected.

Double representing a double precision floating point value. This will accept any floating point number, including in scientific notation.

String for a string of characters. Input strings should not be quoted. When defining schemas, Strings support an extra "enumeration" field. This allows the valid strings to be restricted to a set. In the schema editor this is input as a comma separated list of values. For example, to restrict the input to the set of strings "x", "y", or "z", the input should be x,y,z. This can be surrounded by optional square brackets ([and]), and there can be internal spacing (so $[x,\ y,\ z]$ will also work).

Integer a type representing an integer value. This is strict, so 1.0 is not an integer.

Path this is used for holding a file system path. This is checked when entered, and a warning is issued if the path does not exist.

Array a type representing an array or matrix. The array is further typed with one of the six basic data types and inputs are bound by the input rules of that type. The syntax for Array inputs is designed to be as close as possible to the syntax for MATLAB arrays; hence a comma separated list is used for columns, and rows are separated using a semicolon[18]. For example, a 2×2 matrix would be input as $[1, 2; 3, 4]$.

Component a *slot* for a suitable model which will have its own parameters. The selected model is considered the 'value' of this parameter type. Being a branch parameter, a component will have a branch name.

Struct this acts as a container of child parameters, much like a struct or record in other languages. A struct is the only data type not to have a value, it will however have a branch name.

[18] Unlike MATLAB, the square brackets are optional, and the comma is not.

B.2. Additional Properties

Each data type has a number of additional properties that are stored within the schema or project file. These are used to provide additional restrictions on the values that can be entered (e.g. minimum and maximum values), units and default values.

Table 2 shows which properties are available for each data type.

Table 2: Available properties for each data type

Data type	Default	Units	Restrictions
Boolean	✓	✗	None
Complex	✓	✓	None
Double	✓	✓	Optional minimum and maximum
String	✓	✗	Optional enumerations
Integer	✓	✗	Optional minimum and maximum
Path	✓	✗	None
Array	✓	✓[a]	Base data type, optional min. and max. length
Struct	✗	✗	Duplicable
Component	✗	✗	Model type and duplicable

[a] only if base data type allows units

B.3. XML Data Structure

As discussed in Section 5, the XML schema and project input files are closely related. Every parameter in these files is defined by an XML element. The data type of the parameter is used as the element name, with all other data defined as attributes or child elements. A summary of these attributes grouped by parameter/data type is shown in Table 3.

B.4. Units

The pre-defined unit types and their units are shown in Table 4, the first unit listed is the canonical unit.

Table 3: XML attributes for schema and project files grouped by parameter data type

Attributes	Description
All Parameters	
param_name	The name of the parameter.
description	A short description about the parameter to help the user.
user_description	A comment provided by the user about the entered value. This is empty in the schema file.
Leaf Parameters	
value	The value entered by the user. In the schema this is the default value.
String	
Strings have no additional attributes, but they do have enumeration child elements.	
Bounded numeric data types.	
min_value	If non-empty, the value attribute must be $\geq$ than this.
max_value	If non-empty, the value attribute must be $\leq$ than this.
Unit enabled data types	
units	The unit type
current_unit	The unit selected by the user or default unit. The value is in this unit.
Arrays	
The element name of an array is concatenated with the base data type e.g. DoubleArray. Values are stored in multiple value child elements	
min_len	If non-empty, the number of items in the array must be $\geq$ than this.
max_len	If non-empty, the length of the array must be $\leq$ than this.
Branch Parameters	
user_name	A name given to this branch by the user to uniquely identify it from sibling parameters with the same param_name.
duplicable	Takes value 1 if the parameter may be duplicated, or 0 otherwise.
Components	
model_name	This is the name of the model and is empty in a schema file, except for the root element. In a data file this indicates the model that the user has selected to connect to the component parameter.
model_type	The type of model that may connect to this parameter. In the case of the root element this represents the model type of the defined model.

Table 4: Predefined units in Cards as presented to the user. The first unit listed in each row is the canonical unit

Unit type	Units
angle	deg, rad, millirad, mil (NATO)
length/distance	m, km, ft, nmi
frequency	Hz, kHz, MHz, GHz, rpm
temperature	K, C, F
short time	s, ms, us, ns
long time	s, M, H
speed	m/s
absolute	absolute, dB
extended absolute	absolute, dB, percent, permil, basispoint
power	W, kW, MW, dBW, dBmW
rotation	deg/s
acceleration	m/s2
dB_m	dB_m
reflectivity	m2

B.5. Unit Errors

In the case that an error is detected in a user defined unit addition/extension, the message `ERROR: Some user defined unit conversion(s) failed!` will be logged along with some diagnostic messages. These messages and likely causes are shown in the following paragraphs.

ERROR: <typename> already exists as a unit set!

This occurs if a unit set with *typename* already exists. This can be solved by renaming the new unit type, or extending the existing type (see Section 4.3.2) as appropriate.

ERROR: <unitname> is not in the set of specified units!

This is likely because of a spelling mistake when trying to specify a conversion.

ERROR: No conversion found for <unitname> to <unitname>

This is caused by a missing conversion function. If multiple functions are missing, Cards will only report the first missing function definition.

ERROR: <typename> is not an existing unit set!

This occurs if extending a set that doesn't exist. Check the list of units in Table 4.

Appendix C. Example Code for PROP

C.1. System Definition

Included here is the full code for the *PROP* class introduced in the Section 4.1. To use this system in Cards, prop must be added to the list of imports in *backends/__init__.py*.

```
# Remember to place this in prop.py under the backends folder,
# and to add prop to the list of imports under
# backends/__init__.py

# -*- coding: utf-8 -*-

#
# Defence Science & Technology Group
#
# Unclassified.

from __future__ import absolute_import

from datetime import datetime
import sys
import threading

from cards.utils import override
from cards.backends.system import System
from cards.backends.system_manager import SysManager

@SysManager.register
class PROP(System):
    '''Test Integration System for CARDS. This is an
    example of how to incorporate a new system into CARDS.
    '''

    def __init__(self, project_directory=None):
        self._project_path = project_directory
        self._thread = None

    @property
    def system_name(self):
        return 'PROP'

    @override(System)
    def export(self, root_parametert, directory):
        # Convert input into whatever structure we want
        print('Building PROP Input')
        print('Exporting PROP input to {}'.format(directory))

    @override(System)
    def run(self, system_directory, project_directory, enable_run_signal,
            log_run_signal, stop_signal, finish_func=None):
        stop_signal.triggered.connect(self._stop)
        self._thread = threading.Thread(
            None, self._do_run, None,
            enable_run_signal, log_run_signal)
        self._thread.start()
```

```
    def _stop(self):
        self._thread.terminate()

    def _do_run(self, enable_run_signal, log_run_signal):
        print('Running PROP Model')
        # Perform model run with the inputs that were generated
        print('Emitting log signal')
        log_run_signal.emit(
            'PROP', datetime.now(), 'Dummy path', True)
        # Re-enable the Run menu
        print('Re-enabling Run menu')
        enable_run_signal.emit(False)
```

C.2. Plotting

```
# Import the system backend classes
from cards.backends.prop import PROP

from cards.plot.plot_manager import PlotWrapper
from cards.plot.plot_type import PreProcess


@PlotWrapper(PROP)
class AmbiguityPlot(PreProcess):
    '''Generates a plot of the antenna ambiguity.'''

    def __init__(self, system, root):

        self._extractor = MyExtractor()
        input_data = self._extractor.extract(root)
        hgt = input_data['height']
        freq = input_data['frequency']

        self._generator = MyGenerator(freq)

        self._x, self._y = \
            self._generator.generate_data(hgt)

    def plot(self):
        p = Generate2DPlot()
        p.plot(self._x, self._y,
               x_label='Time delay (us)',
               y_label='Doppler shift (kHz)')
```

In the above, *__init__* is responsible for setting up the Extractor, using it to extract the required data, setting up and passing this data to the Data Generator and saving the data from the generator as instance variables. These are then accessed when `plot` is called to generate the final plot.

```
class MyExtractor(CommonExtractor):
    '''An extractor class that is used to get all of the
    values from the PROP schema that are needed to
    create an ambiguity plot.
    '''

    def __init__(self):
        base_path = 'PROP'
        extras = ['hgt', 'freq']
        wanted = ['height', 'frequency']
        values = {'.'.join((base_path, extra)) : want for extra,
                  want in zip(extras, wanted)}
        super(MyExtractor, self).__init__(**values)
```

The code above is doing only one thing: it is creating a dictionary that maps hierarchical names to the desired names. Effectively, it builds a dictionary that looks like the following:

```
base_path = 'PROP'
values = {base_path + 'hgt' : 'height',
          base_path + 'freq' : 'frequency'}
```

C.3. Schema

Listing 1: Example XML schema file

```
<Component param_name="" duplicable="0" user_name="root" user_description=""
        tag="" description="" model_name="Quick PROP" model_type="PROP">
    <Struct param_name="Refractivity Profile" duplicable="1"user_description=""
            tag="" user_name="Component"description="Profile against height">
        <Double param_name="Range" user_description="" value="0.0" required_unit=""
                min_value="0.0" units="length/distance" tag="" current_unit="m"
                description="Distance from antenna where this profile starts" />
        <DoubleArray param_name="Refrac" user_description="" required_unit=""
                description="The refractivity profile" units="" min_len="2"
                current_unit="" max_len="20">
            <value>130</value>
            <value>130</value>
        </DoubleArray>
        <DoubleArray param_name="hgt" user_description="" required_unit=""
        description="Height of Refrac samples above ground"
        units="length/distance" min_len="2"  max_len="20" current_unit="m">
            <value>1000.0</value>
            <value>1000.0</value>
        </DoubleArray>
    </Struct>
    <Component param_name="Antenna" user_description="" duplicable="0"
            user_name="Component" tag="" model_name="" model_type="PROP_ANTENNA"
            description="The Transmitting antenna pattern" />
    <Double param_name="hgt" user_description="" tag="" required_unit=""
            min_value="0.0" description="Height of the antenna above ground"
            max_value="1000.0" units="length/distance" value="" current_unit="m"/>
    <Double param_name="freq" user_description="" tag="" required_unit=""
            min_value="10.0" description="RF Frequency" max_value="20000.0"
            units="frequency" value="" current_unit="MHz"/>
```

```
    <String param_name="pol" user_description="" description="Polarisation" value="V">
        <enumeration>V</enumeration>
        <enumeration>H</enumeration>
    </String>
</Component>
```

DEFENCE SCIENCE AND TECHNOLOGY GROUP DOCUMENT CONTROL DATA			1. DLM/CAVEAT (OF DOCUMENT)
2. TITLE Cards Reference Manual		3. SECURITY CLASSIFICATION (FOR UNCLASSIFIED LIMITED RELEASE USE (L) NEXT TO DOCUMENT CLASSIFICATION) Document (U) Title (U) Abstract (U)	
4. AUTHORS Daniel Finch, Paul Berry and Zane Van de Meulen-Graaf		5. CORPORATE AUTHOR Defence Science and Technology Group PO Box 1500 Edinburgh, South Australia 5111, Australia	
6a. DST GROUP NUMBER DST-Group–TN–1803	6b. AR NUMBER AR-017-264	6c. TYPE OF REPORT Technical Note	7. DOCUMENT DATE August, 2018
8. OBJECTIVE ID	9. TASK NUMBER AIR 17/514	10. TASK SPONSOR Director General Combat Capability - Air Force	
11. MSTC Surveillance and Reconnaissance Systems		12. STC Surveillance Modelling and Analysis	
13. DOWNGRADING/DELIMITING INSTRUCTIONS `http://dspace.dsto.defence.gov.au/dspace/`		14. RELEASE AUTHORITY Chief, National Security and ISR Division	
15. SECONDARY RELEASE STATEMENT OF THIS DOCUMENT *Approved for Public Release* OVERSEAS ENQUIRIES OUTSIDE STATED LIMITATIONS SHOULD BE REFERRED THROUGH DOCUMENT EXCHANGE, PO BOX 1500, EDINBURGH, SA 5111			
16. DELIBERATE ANNOUNCEMENT No Limitations			
17. CITATION IN OTHER DOCUMENTS No Limitations			
18. RESEARCH LIBRARY THESAURUS Modelling, User manuals			
19. ABSTRACT This document describes the use and design of the software tool Cards. Cards is a user interface for structured parameter inputs, and is easily customised to suit a wide variety of software and hardware systems.			