Running LDDTool and Verifying the Output

Command Line Options for *LDDTool*

The minimal "Operation" documentation provided with the LDDTool package is apparently unmaintained, is at odds with actual tool behavior in more than a few cases, and is flat-out wrong in at least one key point. The information output as help text for LDDTool is not much better. The information below was compiled by running LDDTool with various options and looking at the results produced using LDDTool *version 7.0.1 of the release package (version 0.2.1.0 of the tool itself). The documentation remains unchanged in the latest release.*

Single-letter switches may be combined and order is not significant, so "**-l -p -M**" is equivalent to "**-Mlp**".

Note that the few errors in command invocation that are actually trapped will send error messages to the command line, but these will always be followed by a dump of usage information, so most people will have to be able to scroll back through screen output to see the actual error message.

Also, note that *LDDTool* will silently ignore invalid switches, so type carefully.

Command Switches to Use

These switches seem to operate as described below. Some have both a short form and a long form.

-l	**Required**	This switch must *always* be specified if you want to actually process the input file. Failing to include results in an output list of errors complaining about things that can't be found and nothing else.
-p	**Required**	This switch must *always* be specified if you want to actually process the input file. Omitting results in a a brief error message identifying the missing option followed by a listing of the help text.
-c	**Optional**	This switch directs *LDDTool* to create XML <element> definitions for every *DD_Class* declaration in the *Ingest_LDD* input, ignoring any <element_flag> attributes you might have included in your *DD_Class* definitions. *This is probably not a good idea. Don't do this unless you really mean it.*
-h, --help	**Optional**	This displays the command summary information. In this case all other switches and arguments are ignored. The long form of the switch is safe to use in this case because the **-h** option overrides and cancels all other options.
-J	**Optional**	This switch causes the creation of an additional output file that contains the dictionary information in JSON (Java Script Open Notation) format. It is a dump of the entire Information Model as known to *LDDTool* after processing your input file - so it includes the entire *pds:* core namespace, the known discipline name spaces, and the dictionary just created.

-m	**Optional**	This switch causes *LDDTool* to create an additional output file with the same name as the output schema files and an extension of ".pont". This file is used to load the ontological data base at Engineering Node that holds all configured PDS4 data dictionaries. You should never have to generate this file in normal operations, but if you can find a use for it, have at it.
-M	**Optional**	This switch adds a /mission/ level to the namespace identifier defined for your dictionary. *If you are working on a mission dictionary, you should use this switch to avoid having to edit the namespace in the output schema files.*
-n	**Optional**	This switch adds "nuance property maps" to the output schema and, if any, JSON files. This is an experimental implementation still working towards proof-of-concept stage, so you should avoid it unless you are directly involved in the testing and development of this capability.
-s	**Optional**	This switch causes the version number of the local dictionary to be set equal to the version number of the core namespace used to process the input file, and it also causes the output files to have names of the form *PDS4_ns_vvvv*, where *ns* is the namespace abbreviation specified inside the input file, and *vvvv* is the collapsed, four-digit reference to the PDS core version. Apart from the version number, the output schemas are otherwise unaffected.
-v, --version	**Optional**	This switch causes *LDDTool* to output its internal version number, which is different from the version number on the delivery package. As of this writing, the latest available download package has a version number of "7.0.1", but the **-v** option will report a version number of "0.2.1.0". The long option is also safe to use in this case - the **-v** option overrides all other options except the **-h** option.
-1	**Optional**	This switch (it's a number one) causes the output of the PDS4 Information Model in the same format as the HTML web page on the main PDS site. It will include *attribute* definitions from your local dictionary in "Section 26", but it will not include your *class* definitions - even if you include the **-c** option to create visible class elements, or set the *element_flag* in your class to *true*. Neither do the associated links work. On the whole, there's not much point, but the switch does not appear to have any side-effects on the output schemas themselves, so in that sense it is harmless.

Command Switches to Avoid

Do Not Use These Options. If you read the help text output by *LDDTool*, you will see these switches listed, but they do *not* have the advertised effect(s), and frequently have unintended side effects because of a program error in handling long-form options.

-a	This switch has no effect.

--attribute	This switch has no effect. It is defined as a synonym for the **-a** switch, which has no effect, and it also, coincidentally, has no letters in it that would otherwise be recognized as switches.
--class	This switch is supposed to be an alias for the **-c** switch. While it does cause element definitions to be written for classes, it also changes the name of the output files as though the **-s** switch was included as well.
-d	This undocumented switch causes *LDDTool* to insert *<xs:annotation>* elements into the output XML Schema for every attribute included in each class. These elements contain the human-readable definitions you included in the input file as you defined each attribute. Normally, the output XSD schema file only contains these annotations for the classes. Using this switch causes them to be added for attributes as well. It does not, however, do this in a way that is consistently syntactically valid - so a non-trivial input file is likely to produce an .xsd output file that is riddled with content errors.
-IM Spec	Assuming you are clever enough to figure out how to pass an option with an imbedded blank, this is advertised as a long form of the *-1* option. It is not. It does not trigger creation of the HTML file, but it does have the same effect as specifying the **-M** and **-c** options.
--JASON	This was supposed to be an alternate way of including the **-J** switch. It actually does also trigger the creation of the JSON output file, but this is not how you spell "JSON", and one must assume that it is only the happy circumstance that none of its letters correspond to currently existing other switches that prevents this string from having unexpected side effects similar to those listed for other long versions.
--LDD	This is supposed to be an alternate way of including the required **-l** switch. It is not, and using it throws an error.
--merge	As of this writing, this option has the desired effect of being equivalent to the **-m** option with no additional side-effects. Given the issues with other long options, though, this must be assumed to be a happy coincidence, and it would be better in the long run to avoid *all* long options, even the ones that seem to work, until the underlying problem is solved.
--Mission	This was supposed to be an alternate way of including the **-M** switch. It does produce the desired change in namespace, but also produces the addition effects of the **-s** and **-n** switches.
--nuance	This was supposed to be an alternate way of including the **-n** switch, which you also probably should not use, but in addition to adding the property maps to the output, it also causes the same behavior as the **-c** switch, and defines elements for all your classes.
--PDS4	This is supposed to be an alternate way of including the required **-p** switch. It is not, and using it throws an error.
--sync	This was supposed to be an alternate way of including the **-s** switch. In fact, it has the total effect of the **-s**, **-n**, and **-c** switches combined.

Running *LDDTool*

For the following discussions, we'll use the "IngestLDD_Example_Classes.xml" input file included in the example file (File:LDDTool 1900 examples.zip for IM version 1.9.0.0) set as our model. Of course, you would replace this with your own dictionary input file name in practice.

The simplest invocation of *LDDTool* that a mission dictionary creator should use, and will likely most often use, looks like this:

```
lddtool -lpM IngestLDD_Example_Classes.xml
```

If you are creating a discipline dictionary rather than a mission dictionary, omit the "M"; otherwise make sure you capitalize it (**-m** means something else).

If all goes well, this will send about 40 lines of informational output to your screen (you can redirect it to a file if you like using the usual technique for your operating system). Most of the lines will begin with >>info and list various settings and files referenced by *LDDTool*. Towards the bottom you will see these statements:

```
WARNING  Header:  - New steward has been specified:sbn
WARNING  Header:  - New namespace id has been specified:ex
```

This is normal. The "new steward" will be the value you supplied for the <*steward_id*> attribute in your input file, and the "new namespace id" will the value of <*namespace_id*>.

You will also see various counts for things like attributes and classes. These numbers will likely seems extraordinarily high to you, but they include all the classes and attributes in the entire PDS4 Information Model (which is the context in which *LDDTool* works). As you add your own classes and attributes, you will see these numbers go up - but they will never be small integers.

If there are errors encountered, they will likely be reported near the "WARNING" lines, and similarly labeled with "ERROR" or "WARNING". If your input file was syntactically and schematically valid (and you *did* validate your input file before running *LDDTool*, didn't you?) then the most likely cause for errors here will be typographic - mistyping the name of a class, or using the <*name*> attribute value rather than the <*local_identifier*> value in a <*DD_Association*> reference.

Use other options as you want or need to, of course. You can rename the output files before referencing them in labels, if desired, but in a production environment you might find the **-s** option useful to ensure uniformity in naming. And the **-J** option could be useful if you are working with developers who need JSON support for your dictionary.

Validate That Output!

*It is **extremely** important that you verify that your output schemas are actually valid*.

The *LDDTool* processing cannot detect syntax errors that are specific for the XSD or Schematron environment, and if you accidentally typed the wrong input option you may have invalid output related to that. These situations **will not** be flagged as either errors or warnings in the output listing.

Whatever tool you use to validate PDS4 labels will work for the schemas - so a validating editor or command line tool will do the job. Generally, syntax errors here can and should be corrected in the input file and the schemas regenerated until you get a valid output set. If you encounter a validation failure that *cannot* be resolved by correcting the input, please contact your PDS consultant as soon as possible with the details and a copy of the file(s) that produced the failure.

It is also a good idea to open the XSD schema file, at least, in an editor so you can examine its structure and make sure it corresponds to what you were expecting. An XML-aware editor, for

example, can probably tell you at a glance if you have the right number of "element" definitions (preferably one) in your schema.

The Other Output Files

Running *LDDTool* also produces files in addition to the schema files that are the primary output. Here's a brief summary by file extension (the files will all have the same name as the output schemas):

.csv
This file contains a CSV (comma separated values) table with a summary of the dictionary contents. It does include a column with the description you provided in the input file for all classes and attributes, so you may find this file useful in preparing a human-readable version of the dictionary for reviewers to peruse.

.JSON
This is JSON for the entire PDS4 information model as known to *LDDTool*, and including the new dictionary contents just defined. This file would be on order of 60,000 lines (2.5MB) even before adding any of the new dictionary content.

.txt
This file is the *LDDTool* processing log. You might find helpful information in here if you had *WARNING* or *ERROR* flags in your output listing.

.xml
This is a PDS4 product label file for the schema set just created. You could use this as a basis for a label for the schemas, but it will almost certainly require additional editing for documentation, or for updating filenames if you decide to change the schema file names post-production. This label considers the pair of schemas as a single product.

Editing the Output Files

In general, you should correct the input file or use different *LDDTool* command line options rather than editing the output schemas directly. But there are some cases where direct editing of the output might be in order.

Adding Schematron Rules
If the <*DD_Rule*> class cannot create the Schematron statements you need to perform a specific type of necessary validation, you may have to add them manually. Even if you are a Schematron shaman, do not do this as an alternative to using <*DD_Rule*> when <*DD_Rule*> can do the job, because manual editing of the schema files after generation is a potential giant opportunity for failure in maintenance as the archive ages. If you *do* have to add things manually to the Schematron file, please document what you added and why either in a separate text file that you add to the XML product (that is, modify the output label to point to this new file). This will minimize the risk of the documentation being lost or inadvertently destroyed.

Complex namespace relationships
If you are working on a *very* big and *very* complex mission, and have actually agreed with your lead PDS node that you need to further subdivide your mission namespace into several smaller namespaces, then you may need to manually edit the namespace identifiers in your schemas to add a level below the /mission/ level. There are references to the namespace being defined near the top of both the *.xsd* and *.sch* files that will need to be modified. *Do not do this without approval from your PDS lead node.*

Changing File Names

If you decide to change the names of the output schema files from what was generated by *LDDTool*, you will need to edit the output label file to reflect the new names. In this case no changes are needed in the schema files or in the JSON file, if one was created.

Testing the Schemas

*It is **extremely** important that you test your schemas with one or more labels specifically designed to exercise each and every validation condition you created.*

Testing is critical, and can be particularly tricky with Schematron files involved. You should design one or more labels specifically to check that each validation condition you coded into your input file is being properly checked and flagged. This is not as simple as just making sure that a label that was previously valid is still valid. Consider the case of an attribute that must have one of three permissible values. Here are the cases that should be checked before considering the associated schemas valid:

- If the attribute is required and is not present, an error must be reported.
- If the attribute is present and has a permissible value, no error should be reported.
- If the attribute is present and has a value that is not permissible, an error should be reported.
- If the attribute is optional and is not present, no error should be reported.

It is annoyingly easy, for example, to write a Schematron rule to constrain the value of an optional attribute that will report an error if the attribute is not present - thus effectively requiring the presence of an attribute that you had intended to be optional. Schematron rules added to enforce complex relationships between attributes and classes are prime candidates for unintended consequences like this.

It is even easier to introduce a typographical error into a permissible value that will go unnoticed indefinitely if not detected in testing.

And it is practically guaranteed that if you make regular use of the <*DD_Rule*> class in your input file, that at some point you will mistype something in a <*rule_context*> attribute that will prevent that Schematron rule from ever firing. Unless you specifically test for the error that rule should be flagging, you will never know there is a problem. It is not an error to have a non-existent context (or attribute) referenced in your Schematron file.

If you create your test labels in parallel as you develop your dictionary, the task will seem far less onerous, as well as providing some useful experience using the dictionary you are creating.