PYDOP: A Generic Python Library for Delta-Oriented Programming

Michael Lienhardt

PYDOP: A Generic Python Library for Delta-Oriented Programming

Michael Lienhardt
michael.lienhardt@onera.fr
DTIS, ONERA, Université Paris-Saclay
91120, Palaiseau, France

ABSTRACT

Delta-Oriented Programming (DOP) is a modular and flexible approach to implement Software Product Lines (SPLs). In DOP, an SPL is structured in four parts: the *feature model* describes the configuration space of the SPL; the *base artifact* is an initial artifact; the *deltas* are sequences of transformation operations on an artifact; and the *configuration knowledge* indicates for which configuration and in which order the deltas are applied on the base artifact to generate the variant corresponding to the input configuration. In principle, DOP can thus be used to construct SPL over any artifact that can be transformed. In this paper, we introduce the *pydop* library which implements DOP in python3. This library provides all the necessary structures to build SPLs over any transformable python artifact. Moreover, it provides transformation operations over python classes and modules, thus enabling the possibility to build SPLs over python code as well. Finally, this library also provides a simple *Multi*-SPL mechanism that enable SPL interactions.

CCS CONCEPTS

• **Software and its engineering → Software product lines**.

KEYWORDS

Delta-oriented programming, python library

ACM Reference Format:
Michael Lienhardt. 2023. PYDOP: A Generic Python Library for Delta-Oriented Programming. In *Proceedings of 27th International Systems and Software Product Line Conference (SPLC'23).* ACM, New York, NY, USA, 4 pages. https://doi.org/3579028.3609011

1 INTRODUCTION

A Software Product Line (SPL) is a set of programs, called *variants*, which are generated from a common description and have well documented variability [3]. Delta-Oriented Programming (DOP) [2, 5, 6, 10, 13] is a flexible paradigm to implement Software Product Lines (SPLs). A Delta-Oriented SPL consists of: (*i*) a feature model (FM) describing the functionalities available in the SPL; (*ii*) a *base artifact* which is the starting point of the variant generation process; (*iii*) a set of *delta modules* that expresses modifications to the base artifact, and (*iv*) a *configuration knowledge* (CK) stating the *application order* of the delta (i.e., in which order the delta should be applied), and their *activation conditions* (i.e., for which feature combination each delta would be activated).

As the structure of Delta-Oriented SPLs is artifact-independent, providing a generic DOP framework that could be applied to any kind of artifact would avoid most code duplication and associated bugs during the application of DOP to new artifacts, and allow to manipulate any combination of artifacts in a common SPL, which is impossible with the previous implementations. Seidl et al. [11] implemented such a generic DOP framework, called *DeltaEcore*, on top of the Eclipse Ecore model[1]. This framework helps the user to define a modification language for any model expressed in Ecore, and automatically extends it into a DOP language and generates the corresponding variant generation tools. However, while this framework is very generic, some of its characteristics may limit its usage: (*i*) it requires for the artifact to have an ecore model; (*ii*) it requires for the user to define a modification language, which is not an easy task and can seem superfluous for artifacts that already have modification operations (e.g., statecharts); and (*iii*) it does not really allow to define SPLs over combinations of artifacts, as it generates one DOP implementation per Ecore model.

In this paper, we introduce the open-source python3 library *pydop*[2] that was designed with two main goals: (*i*) being *plug and play*, i.e., one just need to have python3 installed and the pydop library loaded to be able to write DOP product lines and generate artifacts; and (*ii*) integrating seamlessly with existing python artifacts, such as code generation tools like SymPy [9] and statecharts execution libraries like Sismic [8]. This generic library can thus be used by developers to try out SPL Engineering and SPL designs before eventually switching to a more adapted SPL approach. Additionally, pydop implements a simple *Multi-SPL* (MPL) mechanism inspired by [4] that enables SPLs use variants of each other. Finally, pydop is used by a complete redesign of the Fluid Dynamics Simulation tool elsA (http://elsa.onera.fr) that involves more than 2000 features: this community had some requests on the design and manipulation of FMs, which we discuss in this paper.

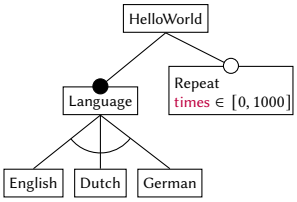
The paper is structured as follows. Section 2 presents the core design of the pydop library; Section 3 discusses the requests raised over the Feature Models; Section 4 describes a toy example using pydop; and Section 5 concludes the paper.

2 PYDOP CORE DESIGN

The core design of the pydop library is to consider the DOP structure as a *Domain Specific Language* (DSL), and like other DSLs (e.g., [1, 7]), to embed it within the python language. This design choice is motivated by four main arguments: (*i*) python is a popular language, and so programmers using pydop will thus most likely code within a familiar environment and syntax; (*ii*) python is a flexible language, which helps implementing a natural syntax to DOP concepts; (*iii*) python is used by many as a scripting language to orchestrate

SPLC'23, August 28–September 1, 2023, Tokyo, Japan
2023. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/3579028.3609011

[1] https://www.eclipse.org/modeling/emf/
[2] available at https://github.com/onera/pydop




```
1 fm = FD("HelloWorld",
2  FDMandatory(
3   FD("Language",
4    FDAlternative(
5     FD("English"), FD("Dutch"),
6     FD("German")
7  ))),
8  FDOptional(
9   FD("Repeat", times=Int(0,1000))
10 ))
```

Figure 1: The HWSPL's FM, with its pydop version

complex and efficient libraries or code generation tools: this means that many artifacts can be constructed and manipulated directly in this language, and consequently with pydop; and (*iv*) python is a dynamic language where it is possible to dynamically create and modify classes and modules: with this functionality, it is thus possible within a single python program to create SPLs over python modules and classes or other artifacts, to generate and use variants of these SPLs.

The rest of the Section presents how pydop encodes the structuring elements of a DOP SPL within python: the FM and the associated configurations, the SPL itself, the deltas, and the CK. We illustrate this encoding using the *Hello World Product Line* (HWPL) [2]. We then conclude this Section with a short discussion about variant generation and the design of Multi-SPL in pydop.

The Feature Model and Configurations. As a first approximation (as previously stated, we will discuss some design choices of the FM in more details in Section 3), the FM is a class-based adaptation of the FM tree structure, as illustrated in Figure 1 which present the FM of the HWSPL and its equivalent in pydop.

`FD` introduces a feature (in order to follow python's syntax, the feature name is a string) that can contain an arbitrary number of groups, attributes and cross-tree constraints (CTCs); `FDMandatory` introduces a mandatory group; `FDOptional` introduces an optional group and `FDAlternative` introduces an alternative group (`Or` groups are also available). The attributes of a features are the named parameters of `FD`, e.g., `times=Int(0,1000)` in line 9 declares the attribute `times` in the feature `Repeat` whose value must be an int between 0 and 1000. CTCs are not illustrated in this simple example. They are stored within the FM tree (in `FD` nodes) and are simple boolean constraints encoded with dedicated python classes: for instance `Implies("Dutch", Leq("times", 3)` means that selecting `"Dutch"` implies that `"times"` must be at most 3. Basic comparisons (e.g., `Leq`, `Lt`, `Eq`) and boolean operators (e.g., `And`, `Or`, `Equiv`) are already defined in pydop, and new classes for more specific purposes can be easily defined.

Configurations and products in pydop are dictionaries mapping feature and attribute names to their values. For instance, the following variable `conf` is a configuration of the FM in Figure 1 selecting the english language, repeated 5 times:

```
1 conf = { "HelloWorld": True, "Language": True, "English": True,
2  "Dutch": False, "German": False, "Repeat": True, "times": 5 }
```

Finally, since a FM can be seen as a predicate over configurations, it is possible to check if a configuration `conf` is a valid product of a FM `fm` by simply applying `fm` on `conf` in `fm(conf)`.

```
1 def gen_base_artifact():
2  res = Module("HW")
3  @add(res)
4  class Greeter(object):
5   def sayHello(self): return "Hello World"
6  return res
7 spl = SPL(fm, RegistryGraph(), gen_base_artifact)
```

Figure 2: The declaration of the HWSPL in pydop

```
1 def Nl(module):
2  @modify(module.Greeter)
3  def sayHello(self): return "Hallo wereld"
4
5 def Rpt(module, product):
6  @modify(module.Greeter)
7  def sayHello(self):
8   tmp_str = self.original()
9   return " ".join(tmp_str for _ in range(product["times"]))
```

Figure 3: The `Nl` and `Rpt` deltas in pydop

The Software Product Line. In pydop, an SPL is simply an object of class `SPL`. The constructor of the class has three parameters: (*i*) the FM of the SPL; (*ii*) the *object managing the order* between the SPL's deltas; and (*iii*) a factory for the base artifact of the SPL. The goal of the second parameter of the SPL constructor is to be generic w.r.t. how the order between delta will be stated. Indeed, several ways to declare this order exist in the literature, and the pydop users community requested to be able to specify this order using the names of the delta. The pydop library provides two classes for managing the order: `RegistryGraph` implements the *after* specification [2] and the global partial ordering specification [4], and `RegistryCategory` implements the naming specification.

The goal of the third parameter of the SPL constructor is to provide the base artifact of the SPL: this factory will be called at the start of every variant generation request, and the result will be used as a basis for the variant generation. That way, every variant generation process will start from a fresh copy of the base artifact.

Example 2.1. One can declare the HWSPL (using the *after* specification for the delta ordering) as in Figure 2. In this Figure, `Module` is the python class for module, and Line 2 creates an empty module named `"HW"`. Like in Example 3 of [2], Lines 3 to 5 add to this module the `Greeter` class configured by default to be in english: the decorator `@add(res)` is a simple transformation operation implemented in pydop that adds the decorated object (here the class `Greeter`) to the parameter of `add`.

The Deltas. Since deltas are executable lists of statements, they must be encoded as functions or methods. For the sake of modularity, we choose to use loose functions or methods to encode deltas, that are later registered to an SPL during the construction of the CK. These functions can take one or two arguments: the first argument is the variant being transformed by the delta, and the optional second argument is the product for which the input variant is being generated. This optional second argument can be useful to query the value of attributes stated in the input product.

Example 2.2. We illustrate how deltas are declared in pydop by implementing features `Dutch` and `Repeat` of Example 2.1 in Figure 3. Feature `Dutch` is implemented with the delta `Nl`. This delta has one argument, the module being modified. The modification code simply

modifies the class `Greeter` of the module (using the `@modify` decorator) by replacing its `sayHello` method.

Feature `Repeat` is implemented with the delta `Rpt`. Here the delta has two arguments: the module being modified like for the `Nl` delta, and the product used to generate the variant. The modification code simply gets from the product the number of repeatitions `nb`, and replaces the `sayHello` method of the class `Greeter`, allowing the original method to be called via `self.original()`.

The Configuration Knowledge. In pydop, the CK performs three tasks on every delta of an SPL: (*i*) it registers the delta to the SPL; (*ii*) it states the activation condition; (*iii*) it states the partial ordering for this delta (this ordering can be implicit for some ordering manager). The activation conditions are written in the same way as the CTCs.

Example 2.3. The CK of our running example is as follows:

```
spl.delta("Dutch")(Nl)
spl.delta("Repeat", after=["Nl"])(Rpt)
```

The first line registers the function `Nl` as a delta of the SPL `spl` declared in Example 2.1, and states that the activation condition of the delta is `"Dutch"` (i.e., this delta is activated iff the feature `Dutch` is selected). The second line registers the function `Rpt` as a delta of the SPL `spl`, states that this delta is activated iff the feature `Repeat` is selected and also states, with `after=["Nl"]`, that this delta must be applied after the delta `Nl`.

Moreover, it is possible to split and integrate the CK with the declaration of the delta using *decorators*. In this case, the declaration of the deltas `Nl` and `Rpt` would look like as follows:

```
@spl.delta("Dutch")
def Nl(module):
 ...
@spl.delta("Repeat", after=["Nl"])
def Rpt(module, product):
 ...
```

Variant Generation and Multi-SPL. Once an SPL is entirely constructed, generating a variant corresponding to a product is done by simply calling the SPL with the product as parameter. The SPL then first checks that the input configuration is a valid product of the SPL, then generate an initial base artifact and iters over all deltas (using the `__iter__` method of the delta ordering object), and apply the activated one on the base artifact. Moreover, since the variant generation process always starts from a fresh copy of the base artifact, it is possible to generate several variants of the same SPL in the same python program.

Example 2.4. In the following, `hw_english` corresponds to the variant of the HWSPL `spl` configured with the English language and no repeat, and `hw_dutch_4` corresponds to the variant of the HWSPL `spl` configured with the Dutch language and 4 repeats

```
hw_english = spl({**conf, "Repeat":False})
hw_dutch_4 = spl({**conf, "English":False, "Dutch":True, "times":4})
```

MPL is a mechanism to construct SPLs that can depend on each other, and in particular the approach described in [4] structures an MPL so that any SPL can access variants of other SPLs in its deltas. As previously discussed, it is possible with pydop to generate any variants of any SPL directly in python, including in functions that are then used as deltas: hence the MPL mechanism comes for free. However, directly generating variants within deltas might lead to generating many times the same variant if it is used in several deltas. To help the user avoiding such variant generation replica, we provide an `MPL` class that acts as an SPL and variant registry and ensures that every requested variant is generated only once.

Example 2.5. The following code snippet illustrates the `MPL` class:

```
1 mpl = MPL()
2 mpl.add("hwspl", spl)
3
4 spl_2_fm = FD("Wrapper", FDOptional(FD("greater")))
5 spl_2 = mspl.new("spl2", spl_2_fm, RegistryGraph(), object)
6 @spl_2.delta("greater")
7 def d2(obj):
8  obj.greater = mpl["hwspl", conf].Greeter()
```

Line 1 declares a new MPL. Line 2 registers the HWSPL in the `mpl` registry, with name `"hwspl"`. Line 5 declares a new SPL directly within `mpl`. For simplicity, this SPL only creates an object. Finally, in Line 8 the delta `d2` accesses the variant of HWSPL identified with the product `conf`, accesses its `Greeter` class and stores an instance of that class in `obj`.

3 FEATURE MODEL AND CONFIGURATION

As stated in the introduction, several requests were raised by users about the FM and configurations.

Multiple Features, One Name. To the best of our knowledge, all formalizations of FMs require for features and attributes to be identified by their name alone. However, in practice, an FM can describe the variability of different parts of a program, each of these parts being related to an existing naming convention: these naming conventions may share names, which is incompatible with the mentioned requirement. For instance in the Fluid Dynamics Simulation tool, several parts of the computation have their own `order` numerical attribute that characterizes the precision of that computation.

In pydop, we allow for several features and attributes to have the same name: they are then identified by *unambiguous partial path*, i.e., any list of names that uniquely identifies the branch where the feature or attribute lives. For instance, in the feature model `FD("elsA", FDMandatory(FD("part1", order=Int(0,3)), FD("part2", order =Int(0,3))))`, the first `order` attribute can be identified with `part1/order`, while the second attribute can be identified with `part2/order`.

Artificially Complex Structure. Structurally, Feature Models correspond to directed hyper-trees, where groups are edges linking one node to multiple nodes. Moreover, while groups model logical connectives (e.g., *mandatory* corresponds to $\wedge$), features also correspond to a $\wedge$ connective over the feature's groups. The elsA developers pointed out this structural complexity of Feature Models, and so we choose in pydop to unify features and groups into one unique class.

Default Configuration and Update. The elsA developers asked for a tool to help writing and updating configurations, while ensuring that no feature would be arbitrarily selected. In particular, completing a configuration using a SAT solver is not acceptable. The pydop library implements this tool with the method `close_configuration` of the class `SPL`. First, the method can complete a configuration in parameter by looking at the SPL's feature tree, adding the mandatory

```python
from pydop.fm_constraint import *
from pydop.fm_diagram import *
from pydop.spl import SPL, RegistryGraph
from pydop.operations.modules import *
import sys

# includes FM from Figure 1, HWSPL from Figure 2 and deltas from Figure 3

def De(module):
 @modify(module.Greater)
 def sayHello(self): return "Hallo Welt"

spl.delta("Dutch")(Nl)
spl.delta("German")(De)
spl.delta("Repeat", after=["Nl", "De"])(Rpt)

conf, err = spl.close_configuration({"English": True, "Repeat": True},
 {"Dutch": True, "times":4})
if(err):
 print(err); sys.exit(-1)

try:
 hw_dutch_4 = spl(conf)
except Exception as e:
 print(e); sys.exit(-1)

print(hw_dutch_4.Greater().sayHello())
```

Figure 4: The Multi-Lingual Hello World Product Line

features and the ones implied by those selected by the user. For instance, the completion of `{"English": True, "Repeat": True}` w.r.t. the FM in Figure 1 is

```
{"HelloWorld": True, "Language": True, "English": True,
 "Dutch": False, "German": False, "Repeat": True}
```

Note that this configuration is not a valid product of the FM because no value was automatically given for the `times` attribute.

Second, the method can update a configuration with modifications wanted by the user: for all updates requested by the user, it updates the configuration by first managing alternatives by unselecting a complete sub-tree of features if it is in conflict with a feature in the update, and then applying a completion pass. For instance, updating `{"English": True, "Repeat": True}` with `{"Dutch": True, "times":4}` w.r.t. the FM in Figure 1 returns the following configuration:

```
{"HelloWorld": True, "Language": True, "English": False,
 "Dutch": True, "German": False, "Repeat": True, "times": 4}
```

4 PYDOP EXAMPLE

In this Section, we complete our running example into a fully executable python program, shown in Figure 4. This Figure starts with a list of imports: Line 1 imports the different classes to construct CTCs and activation constraints; Line 2 imports the different classes to construct FMs; Line 3 imports the classes used to construct an SPL; and Line 4 imports the transformation operations on python objects (including modules and classes). Line 9 presents the `De` delta, which is very similar to the `Nl` delta, and Lines 13 to 15 present an updated version of the CK that includes the `De` delta. Lines 17 to 20 construct a configuration for the SPL and check for possible errors. Lines 22 to 25 generate a variant from that configuration, and check for possible errors. Finally, Line 27 uses that variant by creating a `Greater` object and calling `sayHello()`, which prints:

```
Hallo wereld Hallo wereld Hallo wereld Hallo wereld
```

5 CONCLUSION

In this paper, we presented the *pydop* python library implementing a generic framework for Delta-Oriented Programming that can be used to construct SPL on many artifacts available in the python eco-system. This library is open-source and available at https://github.com/onera/pydop. We motivated the main design choices of the library, and illustrated it with few examples. As future work, we plan to integrate a FM loader and writer compatible with the UVL [12] format, and also to design a syntax for configurations definition. We are also interested to investigate how to implement family-based analysis in this generic framework.

REFERENCES

[1] Marijan Beg, Ryan A. Pepper, and Hans Fangohr. 2017. User interfaces for computational science: A domain specific language for OOMMF embedded in Python. *AIP Advances* 7, 5 (02 2017). 056025.

[2] Dave Clarke, Radu Muschevici, José Proença, Ina Schaefer, and Rudolf Schlatte. 2010. Variability Modelling in the ABS Language. In *FMCO (LNCS, Vol. 6957)*. Springer, 204–224.

[3] P. Clements and L. Northrop. 2001. *Software Product Lines: Practices & Patterns*. Addison Wesley Longman.

[4] Ferruccio Damiani, Reiner Hähnle, Eduard Kamburjan, Michael Lienhardt, and Luca Paolini. 2023. Variability modules. *J. Syst. Softw.* 195 (2023), 111510.

[5] Jonathan Koscielny, Sönke Holthusen, Ina Schaefer, Sandro Schulze, Lorenzo Bettini, and Ferruccio Damiani. 2014. DeltaJ 1.5: delta-oriented programming for Java 1.5. In *Intl. Conf. on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools, PPPJ, Cracow, Poland*. ACM, 63–74.

[6] Michael Lienhardt, Ferruccio Damiani, Lorenzo Testa, and Gianluca Turin. 2018. On checking delta-oriented product lines of statecharts. *Sci. Comput. Program.* 166 (2018), 3–34.

[7] Mathias Louboutin, Michael Lange, Fabio Luporini, Navjot Kukreja, Philipp Witte, Felix Herrmann, Paulius Velesko, and G. Gorman. 2019. Devito (v3.1.0): An embedded domain-specific language for finite differences and geophysical exploration. *Geoscientific Model Development* 12 (03 2019), 1165–1187.

[8] Tom Mens, Alexandre Decan, and Nikolaos I. Spanoudakis. 2019. A method for testing and validating executable statechart models. *Softw. Syst. Model.* 18, 2 (2019), 837–863.

[9] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondrej Certík, Matthew Rocklin, Amit Kumar, Sergiu Ivanov, Jason Keith Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony M. Scopatz. 2016. SymPy: Symbolic computing in Python. *PeerJ Prepr.* 4 (2016), e2083.

[10] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. 2010. Delta-Oriented Programming of Software Product Lines. In *Software Product Lines: Going Beyond (SPLC 2010) (LNCS, Vol. 6287)*. 77–91.

[11] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. 2013. Variability-aware safety analysis using delta component fault diagrams. In *17th International Software Product Line Conference co-located workshops, SPLC 2013 workshops, Tokyo, Japan - August 26 - 30, 2013*. ACM, 2–9.

[12] Chico Sundermann, Kevin Feichtinger, Dominik Engelhardt, Rick Rabiser, and Thomas Thüm. 2021. Yet another textual variability language?: a community effort towards a unified language. In *SPLC '21: 25th ACM International Systems and Software Product Line Conference, Leicester, United Kingdom, September 6-11, 2021, Volume A*. ACM, 136–147.

[13] Tim Winkelmann, Jonathan Koscielny, Christoph Seidl, Sven Schuster, Ferruccio Damiani, and Ina Schaefer. 2016. Parametric DeltaJ 1.5: Propagating Feature Attributes into Implementation Artifacts. In *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2016 (SE 2016), Wien, 23.-26. Februar 2016 (CEUR Workshop Proceedings, Vol. 1559)*. CEUR-WS.org, 40–54.