Layered Programming: A Language Independent Variability Management Approach

J. De Pauw

University of Antwerp
joey.depauw@student.uantwerpen.be

C. Gomes

University of Antwerp
claudio.gomes@uantwerp.be

H. Vangheluwe

McGill University, University of Antwerp , Flanders Make
hv@cs.mcgill.ca

Abstract

Many techniques to implement software product lines exist. Examples are feature-oriented programming, aspect-oriented programming and delta-oriented programming. They are all bound to a specific set of source languages. We propose a way of encoding variability independent of the used language. The goal is to simplify software product line implementation, making it accessible to non-experts. A command line tool is used to achieve this goal, with an optional plugin for FeatureIDE[18, 8].

Keywords: software product line, variability management, feature model, domain implementation

1. Introduction

Layered programming is the concept of writing code in layers. A software product line is represented by a common base and a set of layers. A layer can be seen as an overlay or delta to some program that adds, removes or changes functionality. We use the name "layer" to represent a feature refinement. This term is often used in the context of feature-oriented programming.[3]

It is possible to define a layer on top of another layer, providing a hierarchy. Layers can also be independent of each other. They can then be combined with specific semantics, encoded in a feature model. A Layer is defined by a reference to its base layer and a set of differences with respect to the base.

Note that a layer is not limited to one language or one file. Every system has multiple representations, like source code, makefiles, documentation and so on. Adding a feature to a program should elaborate each of its representations so they are all consistent.[3, 19] An example of this can be found in figure 1. Three layers are defined that modify some source code and keep the documentation consistent. Layer two is defined on top of layer one. Layer one and two are independent of layer three.

Though the name "layered programming" suggests a programming paradigm, it is more closely related to the category of tool support. The technique does not aim to replace the need for a good SPL oriented architecture, but rather to simplify the process of implementing, documenting, organizing and managing the architecture.

Since the technique is language independent, it was tried on an extreme case: the textual representation of models. For this experiment we used AToMPM[17], a tool for multi-paradigm modelling.

In this study, we address following topics:

1. formal definition and workflow description
2. use cases for layered programming
3. implementation outline for a tool to support layered programming
4. analysis of the risks involved
5. effectiveness on textual representation of models

This paper is structured as follows. Section 2 contains the related work. In section 3 a more detailed motivation is given. Sections 4 and 5 propose a working example and explain the solution with layered programming respectively. In section 6 an outline for the implementation is described and section 7 reports the results of the prototype tool developed for this paper along with its integration in FeatureIDE. An application for LP in the modelling field is discussed in section 8. Finally section 9 concludes and section 10 lists future work.

2. Related Work

Birk et. al. investigated SPL practices in the software industry in their paper from 2003[4]. They remark the following about SPL architecture and tool support:

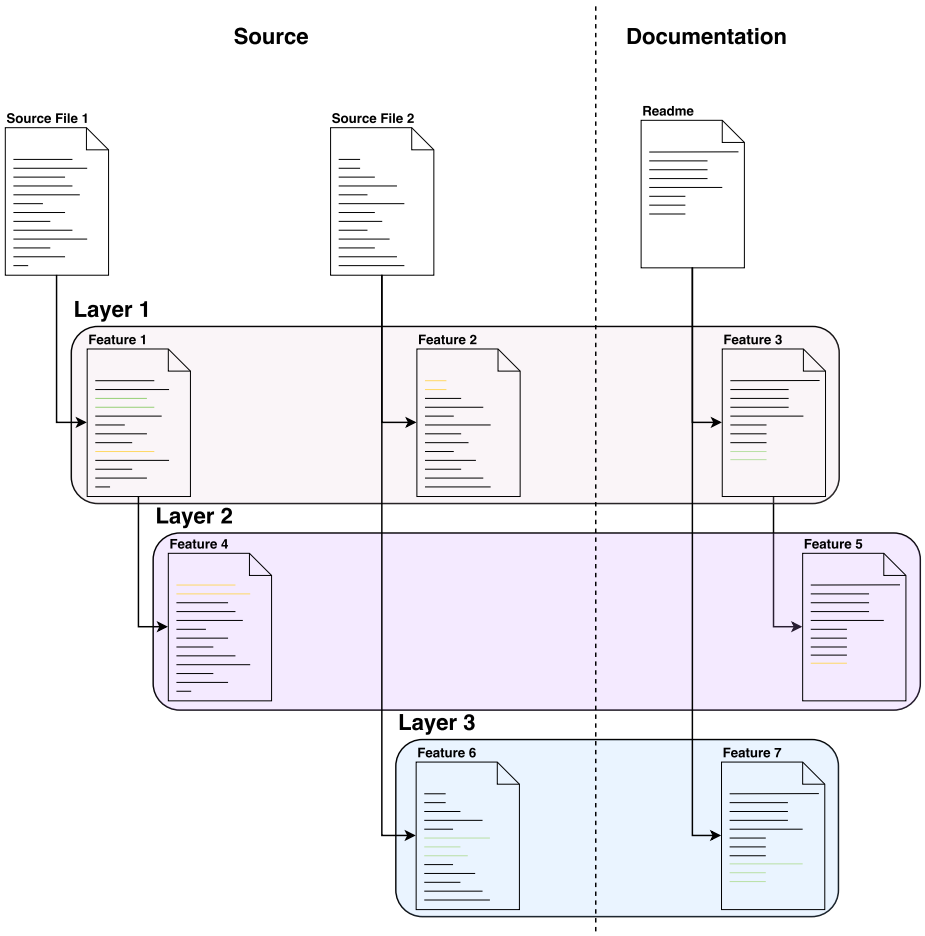

Figure 1: Visual representation of layered programming.

"All products should fit into the provided architecture and benefit from it. Unfortunately, the common architecture's functionality, interfaces, and constraints are usually abstract and complex. Not all the development organization's members or teams will understand them well. Not knowing the SPL architecture's capabilities inevitably leads to the architecture not being fully used."

"Because requirements engineering for SPL can become highly complex, effective tool support is important. Existing tools dont satisfactorily support aspects such as variability management, version management for requirements collections, management of different views on requirements, or dependency modeling and evolution."

Layered programming is a tool-supported technique to assist in SPL development. One of the benefits is that the variability is not hidden away in different files like in most other techniques. Programmers can immediately see all features that affect the piece of code they are working on, preventing code clones and duplicate features.

Since 2003 numerous tools for SPL engineering have been proposed, among which CIDE[7], AHEAD[3], FeatureHouse[1], FeatureIDE[18, 8] and VariantSync[12].

The creation of CIDE was motivated by the problems of both compositional and annotative approaches. It is an Eclipse-based prototype tool for decomposing legacy applications into features that may have a fine granularity[7]. Features are not annotated directly in the code (like preprocessor directives). CIDE manages the feature information and indicates what code belongs to which feature by using different background colors.

The workflow of this tool is closely related to that of layered programming. One of the common advantages is that feature code is still placed where it extends the program, and it is therefore obvious to see how it extends the program, it is simple to understand how a feature is implemented. The main differences are that CIDE uses language specific information (AST representation) to encode features. The CIDE workflow starts with a fully composed application with all features implemented in a single code base, typically a legacy application. It then supports the removal of features to create an artifact. Layered programming implements both bottom up and top down approaches to create software product lines. Artifact are created by adding and combining features, rather than removing them. One of the advantages of this, is that each feature is represented and implemented in the most stripped down version of the source code as possible, minimizing complexity overhead from other features.

AHEAD shows that software can have an elegant, hierarchical mathematical structure that is expressible as nested sets of equations. AHEAD tools are capable of generating Java and non-Java artifacts automatically from nested sets of equations using the Jak-specific tools jampack or mixin.[3, 15] AHEAD uses the Jak language to describe features and to compose them in layers. The technique described in this paper is language independent and does not introduce a new language.

FeatureHouse is a general architecture of software composition supported by a framework and tool chain. FeatureHouse provides facilities for feature composition based on a language-independent model of software artifacts and an

automatic plug-in mechanism for the integration of new artifact languages.[1] This tool generalizes many feature-oriented approaches by providing the typical extend/override mechanism on a language-independent model. It is a descendant of Batorys AHEAD program generator.[1, 3]

FeatureIDE is arguably the most popular open-source framework for feature-oriented software development (FOSD). It is based on Eclipse and supports several FOSD implementation techniques such as feature-oriented programming, aspect-oriented programming, delta-oriented programming, and preprocessors.[18] Numerous other tools have been built on top of the FeatureIDE architecture (e.g.: CIDE, VariantSync, BUT4Reuse).

Shaefer et. al present the idea of delta-oriented programming (DOP) and pure DOP in [13, 14].

Delta-oriented programming is a flexible programming language approach. A product line is represented by a core module and a set of delta modules. The core module provides an implementation of a valid product that can be developed with well-established single application engineering techniques. Delta modules specify changes to be applied to the core module to implement further products by adding, modifying and removing code. A product implementation for a particular feature configuration is generated by applying incrementally all delta modules to the core module.[13]

It relates to layered programming in the sense that it allows a programmer to defined deltas (called layers in this paper) with respect to a core module. Deltas are specified syntactically and grouped together by their functionality. The technique described in this paper is purely tool-based and allows the definition of features right in the core module.

Mezini et. al. address the shortcomings of classic object oriented development in [9]:

"Classes as the traditional units of organization of object oriented software have proved to be insufficient to capture entire features of the software in a modular way. As a result, the last decade has seen quite a number of approaches that concentrate on a more appropriate representation of features in the source code"

They explore language specific solutions like aspect oriented programming and feature-oriented approaches in the context of variability management and investigate their shortcomings. Feature-oriented approaches are defined as a class of approaches that concentrate on encapsulating features as increments over an existing base program, together with a mechanism for combining different features on demand. Existing feature-oriented approaches (FOAs) include: GenVoca [2], mixin layers [15, 16], delegation layers [11], and AHEAD [3].

The technique proposed in this paper can be classified as a language independent feature-oriented approach, not to be confused with feature-oriented programming.

In their paper from 2014, Thüm et. al. conclude that feature-oriented software development (FOSD) provides several techniques for the implementation of SPLs. But, each technique comes with advantages and disadvantages, and that there is no consensus on the best technique.[18]

Like other techniques for domain implementation in SPLs, this one too has its advantages and disadvantages. In section 3 it is compared to existing techniques.

3. Motivation

To overcome the increasing demand for tailored software systems, industrial software development often uses clone-and-own to build a new variant by copying and adapting an existing variant. Indeed, this procedure is easy to use and requires less up-front investments. However, with an increasing number of variants, development becomes redundant and the maintenance effort rapidly grows. Hence, at some point, a sufficient number of variants is reached and the migration to a product line is necessary. However, using a product line to develop variants has several downsides. First, product lines have high up-front investments which make the development of few variants unprofitable. Hence, introducing a product line would be a risky task that could not pay off if the number of required variants is unknown at beginning of development.[12]

To overcome this problem, domain implementation techniques should not only support proactive, but also reactive and extractive product line engineering. In reactive product line engineering, only a basic set of products is developed. When new customer requirements arise, the existing product line is evolved. The extractive approach allows turning a set of existing legacy application into a product line. Development starts with the existing products from which the other products of the product line are derived. [14, 5]

Layered programming can assist in proactive, reactive and extractive product line engineering. Once a feature has been identified, a layer can be derived and extracted from existing code bases. Since this is a purely tool-based technique, there is no need to add code. Depending on the quality of the existing architecture, refactoring may not even be required.

One of the hardest tasks in software product line engineer is complexity management. Often abstract and complex architectural designs are conceived to support variability (e.g.: mixins and mixin layers) or new languages are invented (e.g.: DeltaJ, Caesar, AHEAD). Complexity can even arise from extensive use of preprocessor directives (#ifdef).

Though each of these techniques have their advantages and disadvantages, we can conclude that they all add complexity to a software product line, making domain implementation a task for SPL experts rather than domain or language experts, which is counter intuitive. With the technique presented in this paper, we aim to decrease the threshold for programmers to create software product lines and simplify the transition form a single system to a family of systems.

The most prominent advantages are listed and described.

+ Easy To Use
: With an intuitive GUI, there is virtually no learning curve to LP. There is no new language to learn. No new structures or complex architectures. The only thing that has to be mastered, is the workflow described in section 5.

+ *Semantically Clear*
  LP allows you to define and implement features in the same place you would place them in single system development. This makes features easy to implement, but also easy to find for other developers in the team. By default only the minimal set of required features is visible while working on a feature. This eliminates unneeded complexity from other features. All possible interactions can still be visualized when needed.

+ *Robust*
  This technique tries to keep all features consistent with each other. If a file is modified, all files that depend on it are updated. When this is not possible with high enough accuracy (due to a conflict for example), the error is reported. In some cases, this may even be an indication the developer did something wrong. For example remove a class that a feature still depends on.

+ *Timeless*
  No assumptions are made about the source language. In theory it works and will work for all languages, even those to be created still.

The technique is no silver bullet. There are many scenarios where other techniques outperform LP. These are some use cases where LP in its own does not suffice:

- *Crosscutting*
  Sometimes a feature increment that crosscuts the system is required. It is not trivially possible to achieve this with LP.

- *Large Files With Large Differences*
  The underlying implementation of LP is bound to the size of the input. More so than most other techniques. In particular large files with a lot of differences are harder to support.

- *Runtime Variability*
  There is no support for runtime variability. It can however be used in conjunction with other techniques that provide it.

4. Working Example

A common example used to demonstrate variability in software is that of a Request System (RS).[9] For this purpose, we will consider three files: *Person*, *Database* and *Request*. They are implemented in the *Base* feature. Other features have been defined to interact with them as shown in the feature model in figure 2. An implementation in Python is given in listing 1. The implementation for the database is trivial and hence not included. Features *Balance* and *Age* each add code to the *Person*. Only the interface of the *Database* is defined in *Base*. Either *SQLite* or *MySQL* can be chosen as a back end. Finally there are two features *RequestPricing* and *RequestAgeLimit* that modify how requests are made.

Base/person.py

```
class Person:
    def __init__(self, name):
        self.name = name
```

Base/request.py

```
from database import Database
db = Database("[conn␣string]")

def makeRequest(query, person):
    result = db.execute(query)
    return result
```

Balance/person.py

```
class Person:
    def __init__(self, name):
        self.name = name
        self.balance = 0

    def charge(self, x):
        self.balance -= x

    def deposit(self, x):
        self.balance += x
```

RequestPricing/request.py

```
from database import Database
db = Database("[conn␣string]")

def makeRequest(query, person, price):
    result = db.execute(query)
    person.charge(price)
    return result
```

Age/person.py

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

RequestAgeLimit/request.py

```
from database import Database
db = Database("[conn␣string]")

def makeRequest(query, person):
    if person.age < 18:
        return None

    result = db.execute(query)
    return result
```

Result/person.py

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        self.balance = 0

    def charge(self, x):
        self.balance -= x

    def deposit(self, x):
        self.balance += x
```

Result/person.py

```
from database import Database
db = Database("[conn␣string]")

def makeRequest(query, person, price):
    if person.age < 18:
        return None

    result = db.execute(query)
    person.charge(price)
    return result
```

Listing 1: Source code for RS family.

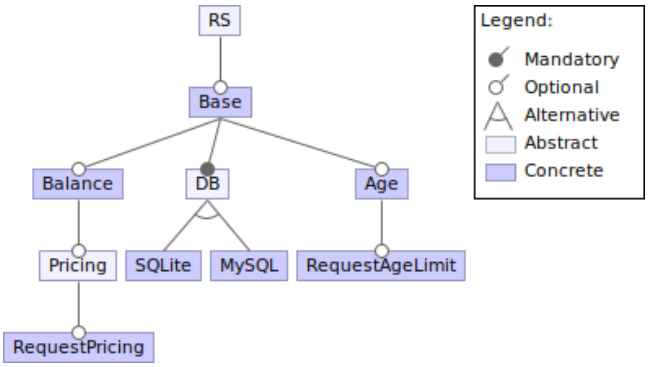

Figure 2: Feature model of example software project.

5. Workflow

This section show how LP can be used for the domain implementation part of SPLE. There are three natural approaches to this: *proactive*, *reactive* and *extractive*. Each of them is explained in its respective subsection.

5.1. Proactive

In this case, the workflow is very straightforward. The preceding domain analysis results in a feature model of all features and their interactions. These can then be implemented top-down starting from the base. Artifacts can be generated by selecting features from the feature model in a configuration and combining them with a layered programming implementation as proposed in section 6.

Unfortunately this methodology can be compared to the waterfall model in software engineering and as stated earlier, it is not applicable in most cases.[12]

5.2. Reactive

A trickier case is reactive SPLE. We start from a basic feature model and want to modify and extend it to support new requirements. Adding features is trivial, since the workflow of proactive SPLE can be followed. Removing a feature, or more realistically a branch of features, is also not an issue. A more complex task is modifying a feature that other features depend on. This can be done by recursively applying the changes made to each of the dependent features where possible, as mentioned in the *robustness* trait in section 3.

Our working example was implemented like this. We started without the *Age* branch and were easily able to include it along the way. Also some changes to the base were made during development. This wasn't an issue because all dependent features updated without a problem.

5.3. Extractive

With extractive SPLE we start from a set of legacy applications (that most likely originated from clone and own practices) and want to create the feature model and implementations per feature. Composing the feature model falls outside the scope of domain implementation and LP, so let's assume that it is possible to construct a FM from the requirements and legacy applications. We then have different versions of the code and we know what features are present in each of them.

From this it is possible to start deriving the features and eventually the base following a simple methodology. Start with one legacy application and extract one feature from it. That is, remove the text that belong to this feature. Then generate a patch that represent the changes needed to add this feature. Continue doing this until only the base is left. Now the minimal sets of code for each feature can be constructed again bottom up by applying the patches as indicated in the feature model. From these minimal code sets, more accurate patches can be generated for future use. This can be done for all legacy applications to cover all features. Different versions of features are best joined manually to select the best parts of each legacy application.

It should be noted that this task is quite fuzzy and error prone. Theoretically it should be possible though.

In the context of our working example, one could have two artifacts: one for the combination {*Balance, RequestPricing, SQLite*} and one for {*MySQL, Age, RequestAgeLimit*}. From these combinations and the feature model it is trivial to extract each feature.

6. Implementation

Up until now, we have focused on what the desired result of layered programming is. This section describes how it can be achieved. Three logical components are required to realize the workflow described in section 5:

- a program for extracting/encoding layers and applying them
- an editor to manipulate different version of the code
- a feature model to describe layer hierarchy and valid combinations of layers

6.1. Program

Two elementary operations are needed to support all the features of layered programming: *extract* and *apply*. Extract takes two files and produces a patch to convert the first file into the second. Apply uses this patch and applies it to a file. These operations closely relate to the *diff* and *patch* algorithms.

Some constraints have to be met on the implementation for *extract* and *apply* operations to ensure a correct result:

- a patch needs to remain applicable under minor, independent changes to its base

- conflicting patches need to be detected and reported, rather than being applied anyways.

The first constraint is clearly needed in the case the base needs to be changed. It is also needed because it has to be possible to apply multiple independent patches consecutively. A degree of fault detection is ensured by the second constraint. Patches are applied in a fuzzy way, based on the context, which is allowed to change. Applying a patch in the wrong place can result in hard to find and hard to fix bugs.

6.2. Editor

Multiple views on the sources are required for software product line implementation. Using LP, there is an instance of the entire project (with only the required layers applied) available for every layer. The layer is edited from this view. Though it is possible to just use existing text editors and the core program to achieve this, an editor proves more efficient, especially for larger projects. This editor does not only show the different views, but also keeps them consistent, which prevents the programmer from forgetting to do this task manually.

A basic prototype of an editor was created as explained in section 7.

6.3. Feature Model

Finally it is clear to see that a way to both describe the hierarchy of layers and interactions between them is required. The feature model formalism is most appropriate for this purpose. Configuration files can also be validated against feature models, providing a way to formally describe different artifacts.

7. Tool

A prototype tool was created to investigate the capabilities of layered programming. This section discusses the development of the core tool and interesting findings concerning its implementation. The core tool was also integrated to FeatureIDE.

7.1. Core Tool

We first tried a possible implementation using the Linux *diff* and *patch* commands. The *diff* command generates a patch file that can be used to turn one file into the other with the *patch* command. Note that the *-u* flag was used to make sure the unified format was used with a number of context lines.

We quickly discovered that the line difference calculated by the *diff* command is not fine grained enough. Layers need to be allowed to make independent changes on the same line like for example adding a parameter to a function.

Another option is to use a character based difference algorithm. Myers et. al. proposed a performant algorithm for this in [10]. It is implemented in Google's

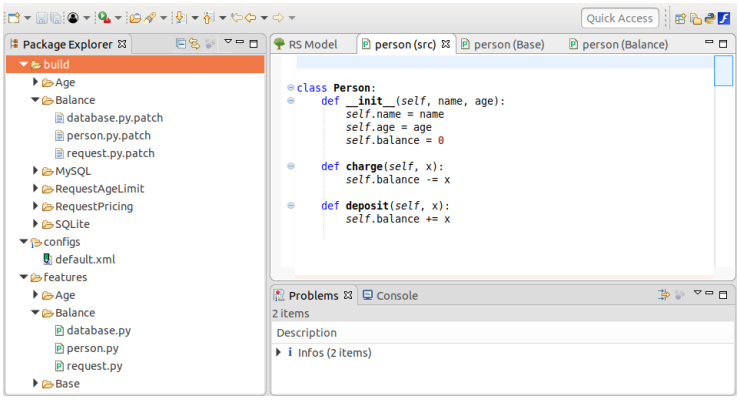

Figure 3: FeatureIDE integration for LP.

diff-patch-match library[6]. However, allowing differences at the individual character level is not necessary for realistic use cases. Dividing the input in words first, proves to be the most effective method. Common separators were used, including spaces, tabs and most special characters like brackets, quotes, etc. Programming languages usually work with tokens over individual characters as well.

7.2. FeatureIDE Integration

This prototype tool was integrated into FeatureIDE[18, 8], an Eclipse plugin for SPLE. It provides a feature model editor with support for creating different configurations. Our plugin used the information encoded in the FM to call the necessary functions of the core tool. A screenshot from the plugin with our working example is included in figure 3.

FeatureIDE has so called "composers" that are responsible for the domain implementation part of SPLE. Existing composers include AHEAD[3], AspectJ and FeatureHouse[1] among others. Layered programming was added using the extension point mechanism of Eclipse. This allows for callbacks when the feature model changed and when an artifact needs to be generated. For most composers, this is enough. We also hooked a change listener to the project files in order for the composer to be notified when a file is updated.

These callbacks are used to generate a minimal view on the project for each feature and keep these views consistent w.r.t. each other during development. For now this is done automatically in the background with very limited user control. One of the benefits of this is that any developer can work on the project. Knowledge about SPLE or experience with variability management techniques is not required. However if anything goes wrong, a global error is reported, but it remains hard to solve without more specific information and more detailed control over the underlying algorithms. Eclipse has a built-in "Compare Editor" that could be useful for this purpose.

The integration to FeatureIDE and Eclipse is not finished by far. Visual indications for modified files, virtual files for storage saving and proper support

for mandatory features are some of the many features that still need to be implemented to make the tool practical.

8. Application: AToMPM Models

The previous sections were motivated by applications in the programming domain. Code has a very rigid structure. Often the order of statements, includes and declarations is important. It's also interesting to look at applications in the modelling field. Models posses a topological structure where order is often less important. AToMPM[17] was used for this experiment.

As an example we chose the Petri Net formalism, of which many variants exist. All of them share the basic idea of places and transitions connected with arcs. Following variants were created in AToMPM:

Capacity
: Adds an optional capacity to places. A place is not allowed to hold more tokens than its capacity.

Color
: Differentiates between types of tokens.

Inhibitor
: Adds an new type of arc from place to transition. This inhibitor arc prevents the transition from firing when tokens are present in the place.

Stochastic
: Refines transitions into *timed* and *immediate* transitions which have a *rate* and *weight* respectively.

For some use cases, a combination of these variants may be required. As a demonstration we combined all of them together. Note that they are not independent of each other. Interactions between variants would normally have to be encoded in another feature and added to the feature model with a constraint. Since we were only interested in combining them syntactically and not in semantics, the interactions were not encoded. The result is a functional combination of the variants, to which no meaning is assigned. Figure 5 shows the abstract syntax model of the base Petri Net formalism and each of the variants as well as the result of combining them.

Each modelling formalism has an abstract syntax and at least one concrete syntax. Layered programming was also applied to the concrete syntax, but without success. By omitting the *stochastic* PN feature, which was the largest, it was possible however. There are two probable causes for this: the size of the models and their encoding. It should be noted that everything is a model in AToMPM and they are stored in Json format.

A lot of redundant information is stored, resulting in very large save files. The largest abstract syntax model had 2468 lines and its concrete syntax counted 8253 lines. It is intuitive to see that it is more difficult for a diff algorithm to handle large files and more important, large differences. Finding the correct location to apply patches evidently becomes harder. Another aspect is the

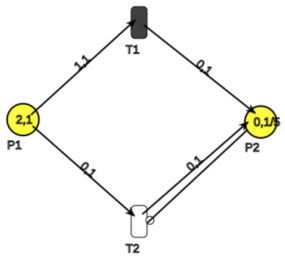

Figure 4: Example Petri Net with all features.

presence of a lot of redundant and hence similar information. Finding the right location to patch is hard when there are a lot of very similar locations.

Users work with model files in AToMPM, but they are compiled to so called metamodel files internally. In essence this strips all information that is not related to the model itself, like the position of a class in the class diagram or the font of each text. Metamodel files are much smaller (224 and 913 lines for the files mentioned earlier respectively).

It was possible for the layered programming prototype tool to combine the compiled versions of the concrete syntax models. This indicates that there is potential in the technique for models as well. It might even be easier because of their topological nature. With an editable version of the abstract syntax and compiled version of a concrete syntax, we were able to use the formalism. An example of this is shown in figure 4.

9. Conclusion

The concept of layered programming was proposed in this exploratory study. Its place in the bigger picture, context, motivation and applications were described. Furthermore a prototype was built and tested on a Python program and a larger modelling example, demonstrating the capabilities and usefulness of the technique, as well as pointing out its flaws and points that require work.

We can conclude that there is potential in the technique, but a lot of risks and uncertainties prevent it from being useful as of yet. The main issues are instability and unpredictability. Further research is needed to see if these can factors can be reduced to an acceptable level by for example providing more user control or more information.

10. Future Work

As mentioned above, better results can potentially be achieved by using the semantics of the source language, rendering the technique only partially language independent. A possible way to do this is by looking at keywords like “class” for example to better analyze the structure of an input. One could also extend the context of patches with a path that leads to the feature, so that it can be fuzzily

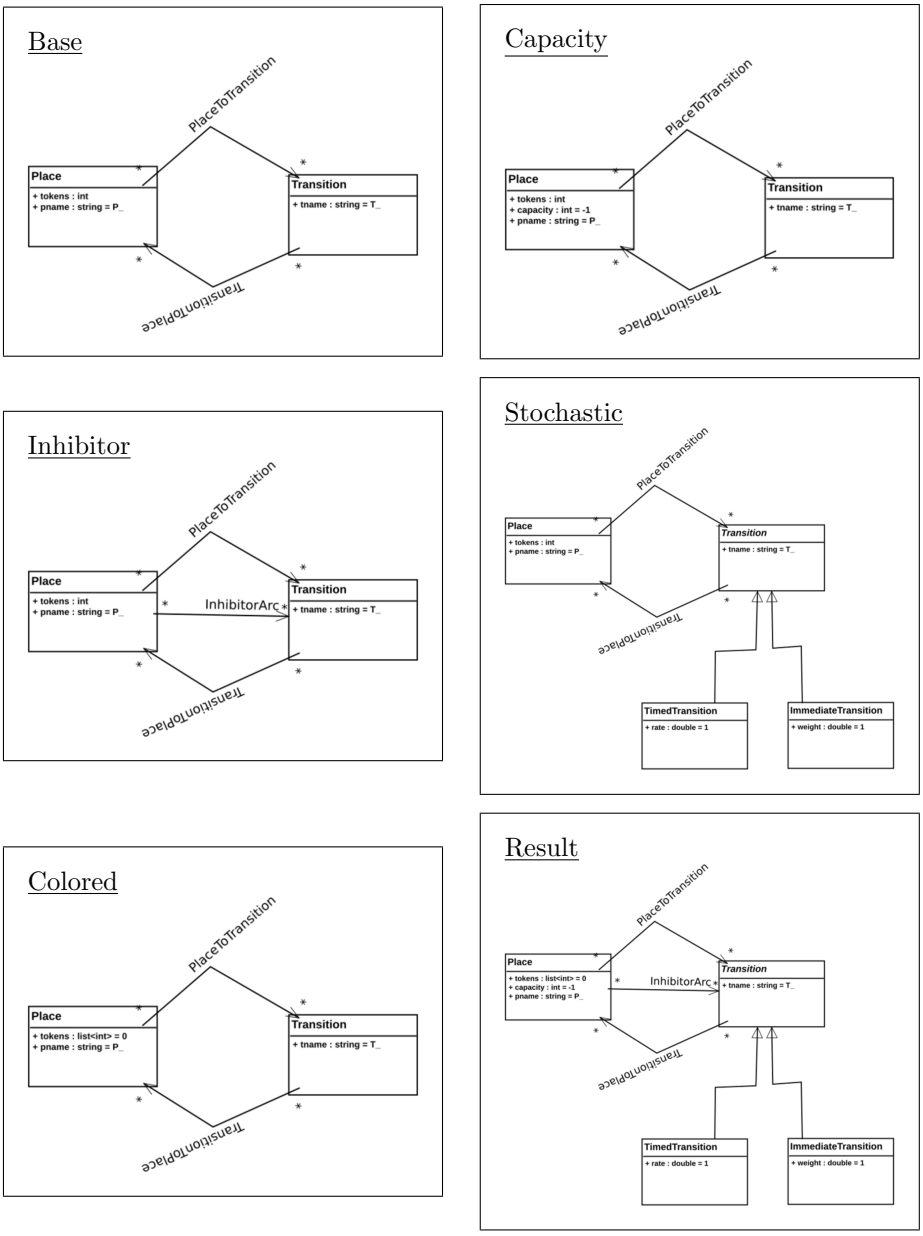

Figure 5: Petri Net models in AToMPM.

applied from there. This path would be language dependent and for example describe the nesting of a piece of code (e.g.: class A: function B: 5th line).

Another open question is the usability of this technique for extractive SPLE. A workflow has been described in section 5, but this has not yet been tried.

There are also disadvantages of encoding variability in patch files. They are not easily readable by users and can't be edited directly. Since these files are source files (features are encoded in them), it should be possible to add them to version control systems. A merge conflict between patch files is difficult to handle properly. A solution for this may be found in merging them on a higher level first by applying the respective patches.

[1] Sven Apel, Christian Kastner, and Christian Lengauer. Featurehouse: Language-independent, automated software composition. In *Proceedings of the 31st International Conference on Software Engineering*, pages 221–231. IEEE Computer Society, 2009.

[2] Don Batory and Sean O'malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(4):355–398, 1992.

[3] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004.

[4] Andreas Birk, Gerald Heller, Isabel John, Klaus Schmid, Thomas von der Maßen, and Klaus Muller. Product line engineering, the state of the practice. *IEEE software*, 20(6):52–60, 2003.

[5] P Clements and CW Krueger. Being proactive pays off/eliminating the adoption barrier. point-counterpoint article in. *IEEE Software*, 2002.

[6] Neil Fraser. google-diff-match-patch-diff, match and patch libraries for plain text, 2012.

[7] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 311–320. IEEE, 2008.

[8] Sebastian Krieter, Marcus Pinnecke, Jacob Krüger, Joshua Sprey, Christopher Sontag, Thomas Thüm, Thomas Leich, and Gunter Saake. Featureide: Empowering third-party developers. In *Proceedings of the 21st International Systems and Software Product Line Conference-Volume B*, pages 42–45. ACM, 2017.

[9] Mira Mezini and Klaus Ostermann. Variability management with feature-oriented programming and aspects. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 127–136. ACM, 2004.

[10] Eugene W Myers. An o (nd) difference algorithm and its variations. *Algorithmica*, 1(1):251–266, 1986.

[11] Klaus Ostermann. Dynamically composable collaborations with delegation layers. In *ECOOP*, volume 2, pages 89–110. Springer, 2002.

[12] Tristan Pfofe, Thomas Thüm, Sandro Schulze, Wolfram Fenske, and Ina Schaefer. Synchronizing software variants with variantsync. In *Proceedings of the 20th International Systems and Software Product Line Conference*, pages 329–332. ACM, 2016.

[13] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. *Software Product Lines: Going Beyond*, pages 77–91, 2010.

[14] Ina Schaefer and Ferruccio Damiani. Pure delta-oriented programming. In *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*, pages 49–56. ACM, 2010.

[15] Yannis Smaragdakis and Don Batory. Implementing layered designs with mixin layers. *ECOOP98Object-Oriented Programming*, pages 550–570, 1998.

[16] Yannis Smaragdakis and Don Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):215–255, 2002.

[17] Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Simon Van Mierlo, and Hüseyin Ergin. Atompm: A web-based modeling environment. In *Demos/Posters/StudentResearch@ MoDELS*, pages 21–25, 2013.

[18] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. Featureide: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 79:70–85, 2014.

[19] Wikipedia. Feature-oriented programming — wikipedia, the free encyclopedia, 2017. [Online; accessed 4-December-2017].