Projectional Language Workbenches as a Foundation for Product Line Engineering

Markus Voelter

independent/itemis
Oetztaler Strasse 38
70327 Stuttgart
voelter@acm.org

Abstract: In this paper I explain the benefits of projectional language workbenches for product line engineering. The ability to extend programming languages with domain specific concepts, mix programs (i.e. descriptions written in general purpose languages) and models (i.e. descriptions expressed with DSLs) and also overlay configurative variability to both of these promises highly integrated and productive tools for product line development.

1 Variability in PLE

In the context of product line engineering, DSLs are used to bind variability. We distinguish between two kinds of variability: structural and non-structural. Structural variability is described using creative construction DSLs, whereas non-structural variability can be described using configuration tools. Structural variability is unbounded, whereas configuration variability is usually bounded. Figure 1 illustrates the spectrum of languages commonly used for expressing and binding variability, reusing an illustration by Krzysztof Czarnecki [6].

Figure 1 Spectrum of Variability

Let's look at two well-known ways of representing variability. Figure 2 shows the meta model of a creative construction DSL. Any number of models can be defined, by instantiating meta model elements. Figure 3 presents an example using the familiar concrete syntax of UML.
Figure 2 A simple meta model for a creative construction language

Figure 3 Two example models

Figure 4 shows a feature model of a weather station using the well-known feature modeling notation [6]. The feature model expresses a certain configuration space, i.e. the model is an expression of configurative variability. A specific configuration is described by selecting a valid subset of those features.

Figure 4 Weather station feature model

Product line development often involves both kinds of variability although there is a bias towards configurative variability in mature product lines. In practice, however, product line development tools need to be able to represent both kinds of variability.

More specifically, it should be possible to define arbitrary modeling languages to express product domain specific abstractions. On top of these, it should be possible to express configurative variations of these domain specific descriptions. In this way configurative and structural variability can be combined exploiting the advantages of each.
2 Projectional Language Workbenches

Language workbenches are tools that allow the creation, extension and integration of separate (general purpose and domain specific) languages in a common editing and validation environment. The most important characteristic of projectional language workbenches [1] is that all text, symbols, and graphics are projected. Projection means that programs are stored as an abstract data structure and users edit a projection, a visual or graphical representation of the data structure. Users never edit text that is subsequently parsed, rather, users always edit a tree directly. Consequently, editing things that look textual and editing things that look graphical happens based on the same technological approach, making the various notational paradigms integratable. Figure 5 shows an example of what's possible with projectional editing based on a Pension workbench developed with Intentional's Domain Workbench [2]. It showcases several different languages, all of them symbolically integrated.

![Figure 5 Screenshots from a pension system built with the Intentional Domain Workbench](image)
3 Handling Structural Variability

The core to handling structural variability is the ability to define new, domain specific languages to express aspects of a product in a product line or to extend existing language with new, domain specific concepts. During domain engineering, the new languages are defined and integrated with other existing languages. In application engineering developers use these languages to efficiently express products. Figure 5 shows this approach in the domain of pension contracts. Using suitable language workbenches, this kind of "language mixing" is straightforward to do - this is what language workbenches are built for. Here are some advantages of the approach, relating to handling structural variability in product line engineering.

As you can see in Figure 5, this system has several sublanguages (mathematical, rule script, tabular). All of these various syntax forms are integrated in one workbench, symbols defined with one language can refer to symbols defined with other languages.

Because no parsing is required, language composition is easily possible (the problems of grammar combination don't exit). Separate languages or language modules can be used together only when necessary. A user only sees those language features that are relevant to his tasks. Language adaptation and extension can be incremental and iterative.

Notations are more flexible than pure ASCII or Unicode. Graphical, semi-graphical and textual notations can be mixed and combined. In our case above, mathematical symbols, rule languages and spreadsheets are mixed.

Because the model can be projected in different ways, it is possible to store "out of band" information in the model (pointers to configuration features, traces back to requirements). The information can be stored in the model and only projected (i.e. “shown in the code”) when the programmer specifically wants to see this information. More generally, the same model can be shown in different (textual and graphical) ways.

The nature of projectional editing means that language extension goes along with IDE extension. So, for the language concepts mentioned above the IDE will provide code completion, error checking and syntax highlighting.

Note also that by representing general purpose programming languages in language workbenches, domain-specific descriptions ("models") and things expressed with general purpose programming languages ("code") can be handled the same way. This makes mixing the two trivial and removes the (arbitrary) distinction between programming and modeling. To illustrate this, Figure 6 shows a version of Java that has been extended to include state machines.
4 Handling Configurative Variability

As mentioned in the previous section it is possible with projectional editors to store information in the model that are out of band with the actual model or program. This is extremely useful for overlaying configurative variability. The following screenshots show an example with MPS [3] that supports configurative variability on top of the state-machine-Java. The screenshot on the left shows a “normal” program (a Java class with an embedded state machine). The middle screenshot shows the code annotated with feature expressions that determine under which conditions the respective elements are present in the system - it represents a product line asset. The screenshot on the right shows the pedestrian variant of the traffic lights: all the non-pedestrian code has been removed.
Note that all three representations are "live", i.e. changes can be made in the programs in each of the projections.

In this prototype the features are simply listed in the model (not shown). Subsequent versions of the feature-annotation facility will connect with a feature modeling to integrate the variable models/code with the overall variability management approach. Adding this functionality is straightforward to do by adding a plugin to MPS that interacts with pure-variants' XML files.

5 Practical Experience

Projectional editing is relatively new. Not too much practical experience exits. Here are some cases. Intentional Software has built several systems using this approach, although to my knowledge none of them uses the overlayed configurative variability. Well publicized is the Pension Workbench illustrated above [4]. JetBrains, developers of MPS, have used the tool to develop their new YouTrack issue management system [5].

Personally, I am currently working on adapting this approach for embedded system development by adding C to MPS and extending it to make it more suitable for embedded systems. Together with Bernhard Merkle (Sick AG), Alexander Bernauer (ETH Zuerich) and Jochen Hiller, I am working on a proof of concept based on C, Osek and Lego Mindstorms.
The following is a screenshot of a Hello World program written with the C implementation in MPS. As you can see, a reasonable subset of C is already available. A generator that creates C code (in text files) as an input to the compiler is available. The program shown below can be compiled and run.

Figure 8 C in MPS

Let us look at some specifics of the code:

1. Resources are the concept corresponding to files. Resources contain modules.
2. There are different presentations modes available, the outline view is shown in the illustration below on the left.
3. Modules act as namespace and scope. They can depend on other modules.
4. A simple testing language is available (this is a separate language module that can be optionally included in programs).
For expressing feature variability, sections of the code can be annotated with feature expressions. The feature expressions are a separate, fully tool-supported expression language. Note that feature expressions can be evaluated and variants can be shown in the editor (see illustration below on the right) and also influence code generation.

A special abstraction for tasks is available. They are translated specific to the target platform. The *onceUponStartup* tasks become the contents of a *main()* function on win32, for other platforms this will be translated differently.

A log statement is available. It is also translated specifically for the platform.

Of course, the variability mechanism illustrated in section 4 based on Java is also available for C in MPS.

### 6 Summary and Outlook

As shown in the above examples, projectional editing, language workbenches, modular languages and the integration of (what's classically called) programming and modeling is a very powerful approach for programming in general. Because of the inherent domain-specificness of product line projects, it is especially suitable for product line development. The ability to express configurative variability on any kind of models or code is very useful.

Future work will address an industrial-strength implementation of the feature-annotation facilities for MPS together with a connection to pure::variants. We will also complete the MPS-implementation of C and use it for a real-world demo of the approach for embedded systems product line engineering.

### References


