Implementing Feature Variability for Models and Code with Projectional Language Workbenches

Markus Voelter

Independent/itemis

Oetztaler Strasse 38,
70327 Stuttgart, Germany
voelter@acm.org

Abstract

Product line engineering deals with managing and implementing the variability among a set of related products. We distinguish between two kinds of variability: configuration and customization. Customization variability can be described using programming language code or creative construction DSLs, whereas configuration variability is described using configuration based approaches, such as feature models. Many product lines have both kinds of variability, and they need to be integrated efficiently. This paper describes an approach for product line engineering using projectional language workbenches. These represent code and models with the same fundamental technology, enabling the mixing of models and code. They make the tight integration between several domain-specific languages possible and simple. Since they can store arbitrary information in models, it is possible to overlay configuration variability over customization variability (i.e. apply feature model-based configuration to code and models). Because of the projectional approach to editing, programs can be shown with or without the dependencies on feature models, they can even be rendered (and edited) for a specific variant. This approach leads to highly integrated and productive tools for product line development. The paper explains the approach, outlines the implementation of a prototype tool based on Jetbrains MPS and illustrates the benefits using a small product line for embedded systems.

Keywords Product Line Engineering, Feature Modeling, Domain-Specific Languages, Language Composition
***Classification*:** D.1.2 Automatic Programming, D.2.11 Software Architectures, D.2.3 Coding Tools and Techniques (Program editors), D.2.6 Programming Environments (Programmer workbench)

FOSD'10, October 10, 2010 Eindhoven, The Netherlands
Copyright © 2010 ACM 978-1-4503-0208-1/10/10... $10.00

1. Introduction

The technical implementation of product line engineering focuses on two main issues: a mapping from the problem space to the solution space as well as the management and implementation of variability. In both contexts, domain specific languages (DSLs), i.e. languages that are custom-built to express specific, limited aspects of a (software) system, can help.

When configuring a product, all variation points defined in the product line have to be bound to a variant. Variation points can be bound at different times (for example, when writing the code, during system initialization, or at runtime). A variation point can also vary in the degree to which it can be configured. Two fundamental kinds of variability can be distinguished: customization and configuration.

When binding a configuration variation point, one among several alternatives is chosen. Feature models [11] are a way to describe the configuration options for a set of variation points as well as the constraints between them (such as "feature *A* cannot be selected together with feature *B"*). The number of alternative configurations may be large, but it is bounded, because only a limited number of valid feature combinations exists.

Customization variability is unbounded. A customization variation point is bound by writing a (potentially very small) program in a (perhaps very specific) language. For example, in a framework, a variation point may require the implementation of a class that implements an interface supplied by the framework, or in a data management application, a variation point may expect a regular expression that validates some data. The regular expression example suggests that it may be a good idea to define a domain-specific language (DSL) to be used to bind the variation point. DSLs usually allow the specification of an unlimited number of programs ("you can always add one more box "), but the nature of the programs is defined by the DSL.

In this paper I show how projectional language workbenches (explained in the next section) can be used for product line development, representing configuration and customization variability in the same environmnet. I argue the benefits of using DSLs to bind customization variability and will briefly show how to define DSLs and how configuration variability can be overlaid over arbitrary languages, general-purpose and domain-specific. I also show

how we can layer several languages on top of each other to enable an effective problem space to solution space mapping. An embedded systems product line will be used as the example, and the tooling will be based on JetBrains MPS [21], an Open Source projectional language workbench.

Section 2 describes the basics of projectional language workbenches in general, and MPS specifically work. Section 3 explains a feature called language annotation that is very useful for configuration variability. Section 4 shows our proof-of-concept, section 5 takes a look at future work. Section 6 puts our contribution in relationship to others', and section 7 contains a brief conclusion.

2. Projectional Language Workbenches and MPS

The term Language Workbench has been coined by Martin Fowler in 2005 [1]. In this article he characterizes it as a tool with the following properties:

- Users can freely define languages which are fully integrated with each other.
- The primary source of information is a persistent abstract representation.
- A DSL is defined in three main parts: schema, editor(s), and generator(s).
- Language users manipulate a DSL through a projectional editor.
- A language workbench can persist incomplete or contradictory information.

Projectional editing implies that all text, symbols, and graphics are projected, well-known from graphical modeling tools (UML, ER, State Charts): the model is stored independent of its concrete syntax, only the model structure is persisted, often using XML or a database. For editing purposes this abstract syntax is projected using graphical shapes. Users perform mouse and keyboard actions tailored to graphical editing to modify the abstract model structure directly. While the concrete syntax of the model does not have to be stored because it is specified as part of language definition and hence known by the projection engine, graphical modeling tools usually also store information about the visual layout.

Projectional editing can also be used for a syntax that is textual or semi-graphical (mathematical notations, for example). However, since the projection looks like text, users expect interaction patterns and gestures known from "real text" to work (such as cursor movements, inserting/deleting characters, rearranging text, selection). A projectional editor has to "simulate" these interaction patterns to be usable.

The following list shows the benefits of the approach:

- In projectional editing, no grammar or parser is used. Editing directly changes the program structure (AST). Thus, projectional editors can handle unparseable code. Language composition is easily possible, because composed languages cannot result in ambiguous grammars, a significant issue in classical parser-based systems.
- Notations are more flexible than ASCII/ANSI/Unicode. Graphical, semi-graphical and textual notations can be mixed and combined. For example, a graphical tool for editing state machines can embed a textual expression language for editing the guard conditions on transitions.
- Projectional languages by definition need an IDE for editing (it has to do the projection!), so language definition and extension always implies IDE definition and extension. The IDE will provide code completion, error checking and syntax highlighting for all languages, even when they are combined.
- Because the model is stored independent of its concrete notation, it is possible to represent the same model in different ways simply by providing several projections. Different viewpoints [23] of the overall program can be stored in one model; editing can be viewpoint or aspect specific. It is also possible to store out-of-band data, i.e. annotations on the core model/program. Examples of this include documentation, pointers to requirements (traceability) [5] or feature dependencies [6] - as we will describe below.

As a side effect, language workbenches deliver on the promise of removing the distinction between what is traditionally called programming and what is traditionally called modeling. This distinction is arbitrary: developers want to express different concerns of software systems with abstractions and notations suitable to that particular concern, formally enough for automatic processing or translation, and with good IDE support. Projectional language workbenches deliver on this goal in an integrated, consistent and productive way. They do this by applying the technology known from modeling tools (projection) to editing any notation.

The Jetbrains Meta Programming System

JetBrains' Meta Programming System is an open source projectional language workbench [21]. Defining a language starts by defining the abstract syntax, the editor for the language concepts is specified in a second step. Lastly the generator is defined. It outputs text (for a low-level language) or it transforms higher-level code into code expressed in lower level languages. The higher-level to lower-level generators are not text generators, they transform abstract syntax trees.

Editing the tree as opposed to "real text" needs some accustomization. Without specific adaptations, every program element has to be selected from a drop-down list and "instantiated". However, MPS provides editor customizations to enable editing that resembles modern IDEs that use automatically expanding code templates. In some cases though, the tree shines through: Consider changing a statement like *int i = j+k*; to *int i = (j+k)*2*; you cannot simply move the cursor to the left of *j* and insert a left parenthesis. Rather, you have to select the + operator (the root node of the expression on the right) and use a *Surround with Parens* refactoring. Using (as the hotkey for this refactoring creates an editing experience very similar to "real" text).

Language Definition with MPS

I have described language creation, extension and composition in MPS in a separate paper [22]. This section shows an example as a short summary. MPS, like other language workbenches, comes with a set of DSLs for language definition, a separate DSL for each language aspect. Language aspects include structure, editor, type system, generator as well as support for features such as quick fixes or refactorings.

Defining a new language starts by defining the language structure (aka meta model). This is very much like object oriented programming as language elements are represented as concepts that have properties, children and references to other concepts. The second step is the editor for the language concepts. An editor defines how the syntax for the concepts should look like - it constitutes the projection rules. Figure 1 is an example.

```
editor for concept LocalVariableDeclaration
  node cell layout:
    [- % type % { name } ? = ?% init % ; -]
```

Figure 1. Defining an editor for a local variable declaration statement (as in int i = 2*2;)

Next is the definition of the type system. For example, the type property of a *LocalVariableDeclaration* must be compatible with the type of its *init* expression.

At this point, the definition of the language and the basic editor, as well as the type system are finished. However, to use the new *LocalVariableDeclaration* statement, the user has to bring up the code completion menu in the editor, select the concept *LocalVariableDeclaration* and use tab or the mouse to fill in the various properties (*type, name, init*). A couple of editor customizations are necessary to make sure users can "just type" the declaration. I refer to [22] for details on how this works.

Language Modularization and Extension

I referred above to the ability to modularize and compose languages as a way of breaking down monolithic languages into manageable modules that can be combined as needed. I also already alluded to the relationship between object oriented programming and language definition in MPS. This analogy also holds for language extension and specialization. Concepts can extend other concepts, and subconcepts can be used polymorphically. Languages can extend other languages, too, and the sublanguage can contain subconcepts of concepts in the base language or can override the translation rules (generators) of concepts defined in the base language. Concept interfaces are also available. Using the *Adapter* pattern [29], unrelated concepts can be made to fit together. To use a *B* in places where an *A* (or subtypes) is expected, an adapter *BAdapter* that extends *A* and contains or references a *B* is necessary. As shown in [22], this approach supports embedding of completely unrelated languages.

Languages also define translation rules to lower-level languages or to text. MPS includes an incremental translation engine that reduces program code as long as translation rules are available for the program elements. At the end, text generators output regular program text that can be fed into a compiler.

The language modularization and composition features are extremely useful for handling customization variability, because DSLs that describe a specific aspect of the overall system can be easily integrated with the languages used to implement the rest of the system.

3. Language Annotations

With MPS it is possible to add additional data to program elements that has not been "planned for" when designing the original language. It is possible for language *X* to contribute properties to elements of language *Y* without invasively changing language *Y*. This is a little bit like introductions in aspect oriented programming [2]. It is an extremely interesting feature for adding feature dependencies, i.e. as a way of implement configuration variability. Section 4 shows how this feature is used in the context of embedded systems development.

Defining an annotation for feature dependencies

As figure 2 shows, an annotation declares which elements it targets (*A* in the example). As a consequence, this element (and all its subtypes) appears to possess the additional child *r* declared by the annotation. The child can be used like any other child of *A*. The annotation can reside in a different language than the target elements, supporting external, a-posteriori non-invasive extension of languages.

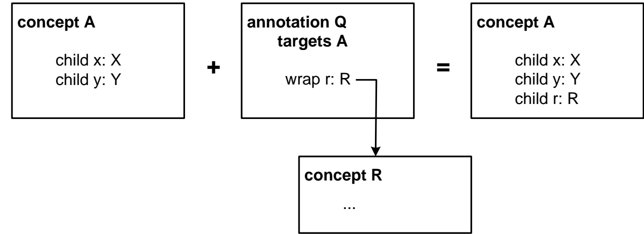

Figure 2. Annotations can add properties to concepts without invasive modification

It is, however, not enough to add the additional property to the program elements. It also has to be rendered in the editor, so the editor needs to be adapted as well. For example, if a program element has a feature dependency annotation, the actual feature expression must be shown near the element. Figure 3 shows an example of three variables that are annotated with a dependency to the *sonar* feature.

```
{sonar} var int currentSonar = 0;

{sonar} var int[10] sonarHistory = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

{sonar} var int sonarIndex = 0;
```

Figure 3. Feature dependencies annotated to variable declarations

This is a placeholder cell which, at runtime, is replaced with the editor of the node to which the annotation has been added. So, in essence, it means that the editor of the annotation element *R* wraps around and embeds the editor to which the annotation has been added.

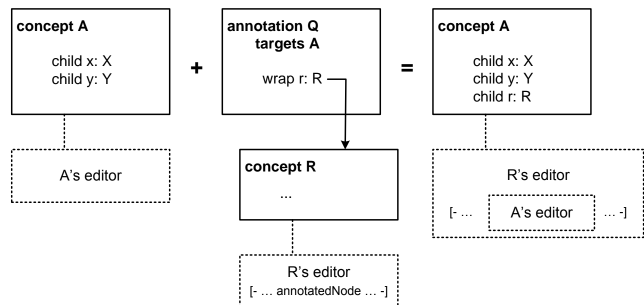

Figure 4. The editor of the annotation "wraps around" the editor of the annotated element

Figure 4 shows how editor annotation works in principle. In the definition of the editor for the element that is "added" to the target, you can use a special editor cell type (*annotatedNode*).

Returning to the example for feature dependencies, figure 5 is the definition of the annotation. It contributes a child named *featureDependencyAnnotation* of type *FeatureClause* to *BaseConcept* and its subtypes (all language element extend *BaseConcept*, at least indirectly. It is like *java.lang.Object*).

```
annotation link declaration featureDependencyAnnotation

    stereotype  node
    cardinality 1
    source      BaseConcept
    target      FeatureClause
```

Figure 5. Definition of an Annotation that adds *FeatureClause* instances to instances of any subtype of *BaseConcept*

In terms of the editor, this means that the editor of *FeatureClause* has to "wrap around" the editor of whichever other element it is contributed to. Figure 6 shows the definition of the editor for *FeatureClause:* it first includes the feature expression (such as *car && !pedestrian*) and then delegates to the node to which it has been annotated using the *attributedNode* element.

```
editor for concept FeatureClause
  node cell layout:
    [- ?[- [- ?[- { % expression % } -] [> attributed node <] -] -] -]
```

Figure 6. Definition of the annotation's editor. Note the delegation to the editor of the annotated node

Evaluating the feature expression during projection

With the facilities described above it is possible to annotate arbitrary program elements with feature clauses. These contain a boolean expression over a set of features. Because the language that is used to define the feature expression is formally defined, it is possible to evaluate the expression in the IDE and show the program in a variant-specific way. This is done by not projecting those elements whose feature clause evaluates to false considering the current feature selection. The projection rules in the editor definition contain the respective *if* statements. Conditional projection is supported by the cells with a leading question mark (see figure 6). These are conditional cells, i.e. they are only shown in the editor at runtime if their associated condition is true. For the feature-aware code editor, the feature annotations themselves (i.e. the *{car && !pedestrian}*) are only shown if a global configuration flag *Show Feature Annotations* is true. So the condition in the conditional cell directly before the *{expression}* in the editor definition in figure 6 contains the following expression shown in figure 7:

```
show if
(scope, editorContext, node)->boolean {
  ConigurationAccessHelper.config(node).showFeatureClause;
}
```

Figure 7. Expression that makes sure the feature clause is only shown if the configuration allows it

```
show if
(scope, editorContext, node)->boolean {
  if (ConigurationAccessHelper.config(node).removeNonSelectedStuffInEditor) {
    return node.isTrue(true);
  }
  true;
}
```

Figure 8. Expression that makes sure the whole annotated element is only shown if the configuration allows it

Also, in the projection mode that shows a program variant, we want to make sure that the program element (e.g. a procedure, or an *if* statement) to which a feature expression is annotated is not shown, if the feature expression is *false*. This is achieved by the outer conditional cell in figure 6 that surrounds the expression and the attributed node. The condition is shown in figure 8.

4. The Proof of Concept

Together with Bernhard Merkle, the author is currently working on a modular language for embedded development based on C (MEL - Modular Embedded Language). It is described in detail in another paper [3]. Among other things it contains support for product line variability as described in the previous section. This section is a brief overview of the language.

Embedded Software Development Language

Embedded systems are becoming more and more software intensive. Consequently, software development plays an increasingly important part in embedded system development, and the software becomes bigger and more complex. Traditional embedded system development approaches use a variety of tools for various aspects of the system, making tool integration a major headache. Some of the specific problems of embedded software development include the limited capability for meaningful abstraction in C, some of C's "dangerous" features (leading to various coding conventions such as Misra-C [4]), the proprietary and closed nature of modeling tools, the integration of models and code, traceability to requirements [5], long build times as well as the consistent implementation of product line variability [6].

To address these issues, we propose a modular modeling and programming language and IDE that supports higher-level abstractions and system-specific extensions based on a projectional language workbench and to use code generation to C as a way of integrating with existing compilers and platforms. The proposed language uses C as its core and adds several useful extensions, including a module system with visibility rules, physical quantities (as opposed to just *ints* and *floats*), first-class state machines, dataflow ports, mathematical notations, memory mapping and bit fields, as well as first-class support for various inter-process communication approaches (shared memory, message passing, bus communication).

As a proof of concept, we are currently building a first cut of this modular embedded language (MEL) based on JetBrains MPS. We use Lego Mindstorms [7] as the target platform together with the OSEK [30, 8] operating system. C and OSEK are widely used in automotive systems, so the technologies used in the prototype are relevant in real systems. The current baseline showcase is a simple line follower robot. It uses a single light sensor to follow a thick black line. It keeps track of the curving line by changing the speed of motors that drive the two wheels. The current state of the prototype contains language modules for components, tasks, state machines, bit-level data structures, physical quantities, documentation annotations, basically all of C as well as support for product line variability and requirements traces.

Configuration Variability - Feature Annotations

Lego is a good way of showing product line variability because it is easy to clip on variant specific hardware. The following two optional hardware elements are available:

- a bumper at the front of the robot that stops it if the bumper is pressed. Essentially, this is a collision sensor.
- a sonar sensor, that temporarily stops the robot if something steps into its way. This is a collision prevention system.

In Figure 9, configuration A shows the robot in its basic setup (*bumper = false* and *sonar = false*), B shows the configuration with the bumper and C shows bumper and sonar.

Figure 9. Three different variants of the robot

Feature dependencies are a way to implement configuration variability in programs. Handling configuration variability requires two ingredients. First, a set of variation points (aka features in the feature modeling approach [11]) and the constraints among them have to be defined. Second, program elements have to be annotated with expressions over these features. These expressions determine whether a piece of program is in a variant or not.

In a real-life system, feature management happens in an external tool (such as pure::variants [12]). In the proof-of-concept, just like with the requirements, features are managed in a simple list. Features can be selected (see figure 9) to determine whether they are in the system or not (which will become relevant later).

The second ingredient are feature annotations, i.e. annotations on program elements that contain an expression that determines which features it depends on, and how. As described above, programs can be projected in a way that shows the feature annotations directly on the element it is attached to (Figure 10). Although the figure contains only dependencies on single features, we can use boolean expressions in the feature annotation such as *{bumper && sonar && !debugOutput}*. This is actually a small sublanguage for boolean expressions (again with code completion into the feature model, error checking etc.).

```
Dummy Feature Model
feature runtimeCalibration : false
feature bumper : true
feature sonar : false
feature debugOutput : true
```

Figure 9. The dummy feature model for the line follower robot (also contains *true/false* switches to define a variant)

As can be seen from Figure 10, feature dependency expressions can be annotated to any program element. Annotated elements have a grey background and the feature annotation expression is given on the left of the element. Alternatively it would also be possible to assign a specific color to each feature and then use the respective color as the background for the elements annotated with this feature (as done by CIDE [19]).

By flipping a switch in the overall projection settings it is possible to show the program in a variant-specific way. For example, if we switch off the *debugOutput* and the *bumper* feature, the code in Figure 11 will result.

```
{sonar} task sonartask cyclic prio = 2 every = 100 {
          int s = ecrobot_get_sonar_sensor(SENSOR_PORT_T::NXT_PORT_S2);
          sonarHistory[sonarIndex] = s;
          sonarIndex = sonarIndex + 1;
          if ( sonarIndex == 10 ) {
            sonarIndex = 0;
          }
          int ss = 0;
          for ( int i = 0; i < 10; i = i + 1; ) {
            ss = ss + sonarHistory[i];
          }
          currentSonar = ss / 10;
          { {debugOutput} { debugInt(2, "sonar:", currentSonar); } }
        }

doc This is the cyclic task that is called every 1ms to do the actual control of the
task run cyclic prio = 2 every = 2 {
  stateswitch linefollower
    state running
      {bumper} int bump = ecrobot_get_touch_sensor(SENSOR_PORT_T::NXT_PORT_S3);
      {bumper} if ( bump == 1 ) {
        {debugOutput} { debugString(3, "bump:", "BUMP!"); }
        event linefollower:bumped
        terminate;
      }
      {sonar} if ( currentSonar < 150 ) {
        event linefollower:blocked
        terminate;
      }
      int light = ecrobot_get_light_sensor(SENSOR_PORT_T::NXT_PORT_S1);
      if ( light < ( WHITE + BLACK ) / 2 ) {
        updateMotorSettings(SLOW, FAST);
      } else {
        updateMotorSettings(FAST, SLOW);
      }
      {debugOutput} { debugInt(4, "light:", light); }
    {sonar} state paused
              updateMotorSettings(0, 0);
              if ( currentSonar < 255 ) {
                event linefollower:unblocked
              }
    {bumper} state crash
               updateMotorSettings(0, 0);
    default
      <noop>;
}
```

Figure 10. A piece of the overall linefollower program projected with feature annotations

Note that these projections still contain the grey highlight for parts that are feature dependent. This can be turned off, too. Also, the feature dependency expressions can be shown in this view if requested. It is important to point out that this is not a read-only projection! Rather, the program can still be edited while shown in the variant-specific way.

As mentioned above, for compilation the program is generated into plain C and then compiled with the existing legacy compiler infrastructure. When generating C, the current feature configuration is taken into account. A simple transformation script is run as part of the incremental reduction process that removes all program elements whose feature clause evaluates to false, making sure they are not part of the resulting C text.

Static Validation of Feature Dependencies

Making parts of models or code optional runs the risk of producing structurally or semantically broken programs after "cutting away" all the stuff that is not configured to be in a certain variant. Detecting semantic errors in turing-complete programs is impossible in general, of course. But static correctness can be verified.

```
task sonartask cyclic prio = 2 every = 100 {
  ...
}

doc This is the cyclic task that is called every 1ms to do the actual co
task run cyclic prio = 2 every = 2 {
  stateswitch linefollower
    state running
      if ( currentSonar < 150 ) {
        event linefollower:blocked
        terminate;
      }
      int light = ecrobot_get_light_sensor(SENSOR_PORT_T::NXT_PORT_S1);
      if ( light < ( WHITE + BLACK ) / 2 ) {
        updateMotorSettings(SLOW, FAST);
      } else {
        updateMotorSettings(FAST, SLOW);
      }
    state paused
      updateMotorSettings(0, 0);
      if ( currentSonar < 255 ) {
        event linefollower:unblocked
      }
    default
      <noop>;
}
```

Figure 11. A part of the program with *debugOutput* and *bumper* switched off (pls compare with Figure 10)

Consider that in MPS (and in projectional editors in general) every element is a node with a unique identity. Relationships between elements are expressed with actual references to these unique identities. A structurally broken program is one where a referencing element is in the code for a given variant, but the reference target is not. Static validation of feature dependencies requires showing that for any (valid) feature combination, no such "dangling pointer" will result. The following is a simple approach to verify this:

- Calculate all combinations of all features (i.e. all variants)
- For each referencing element R, collect all feature combinations C_R for which this element will be in the variant code
- For each reference target element T of R, collect all combinations C_T for which this element will be in the variant code
- If C_R is not a subset of C_T, an error has been detected

This algorithm has been implemented in the prototype and it works well in principle. The fact that all references can be followed easily, and the fact that feature dependencies are expressed as expressions based on a formal expression language makes implementing this algorithm simple - and it does work for small sets of configuration features. But of course the set of possible feature combinations grows exponentially over the number of features, so for real-world sized systems it will not work. The following steps could be taken to address this:

- In real systems, the set of features is not unrestricted, they have constraints among each other. This limits the size of the set of valid feature combinations (i.e. variants)
- Feature macros can be introduced, i.e. features that encapsulate a set of other features and their constraints (e.g. *fast := !small && !dynamic*). If feature dependencies refer to the macro features, they can be seen as *one* combination and the combinatorics behind them can be ignored.
- If only a part of a program needs to be validated, only the feature combinations involving the features referenced from the respective part of the program need to be calculated.
- Finally, using a solver instead of the try-all-combinations brute force approach may yield even more scalable results.

We will explore these alternatives as part of our in the future work (see below).

```
module impl imports <<imports>> {

  int speed( int val ) {
    return 2 * val;
  }

  robot script stopAndGo
    block main on bump block retreat on bump <no bumpReaction>
                                stop
                                accelerate to 0 - 30  within 2000
                                drive on for 2000
                                decelerate to 0  within 1000
                                stop
      accelerate to speed(25) within 3000
      drive on for 2000
      turn left for 2000
      block driveMore on bump <no bumpReaction>
        accelerate to 80  within 2000
        turn right for 3000
      decelerate to 0  within 3000
      stop

}
```

Figure 12. A simple robot routing script

Customization Variability - a DSL on top

Consider now a robot vendor who sells Lego robots with two wheels that can follow a predefined route. Of course, each customer wants a different predefined route. The vendor has to develop a different route-driving program for each customer. Of course this can be achieved with tasks, state machines, variables and procedures, or in other words, the general-purpose MEL. But it would be better if a domain specific language for defining routes was available. In PLE terminology, the DSL would be used to express the problem domain and a transformation would map this to a solution domain implementation.

The program in figure 12 is an example expressed with such as route definition DSL, it uses native route-definition constructs. Since the robot routing language extends the core language, it can be embedded in a module - like the general purpose MEL constructs. The robot script can even call procedures. The robot routing language is executed by transformation into the following MEL constructs:

- a state machine that keeps track of the current command/step
- module variables that remember the current speeds for the two motors
- module variables that store the current deltas for the speeds of both motors to be able to "ramp up" the speeds in accelerate and decelerate commands
- a cyclic tasks that updates motor speeds based on the currently selected deltas.

Figure 13 shows a robot script together with the lower-level program that results from the transformation.

Combining the DSL and Feature Annotations

It useful to combine customization and confguration variability. In the example this would mean that we can attach feature expressions to robot script programs. This is of course also possible. The feature annotations are completely generic and make no assumption about the language to which they are attached. Consequently, they can be used with the robot DSL in the same way as with the lower level programs.

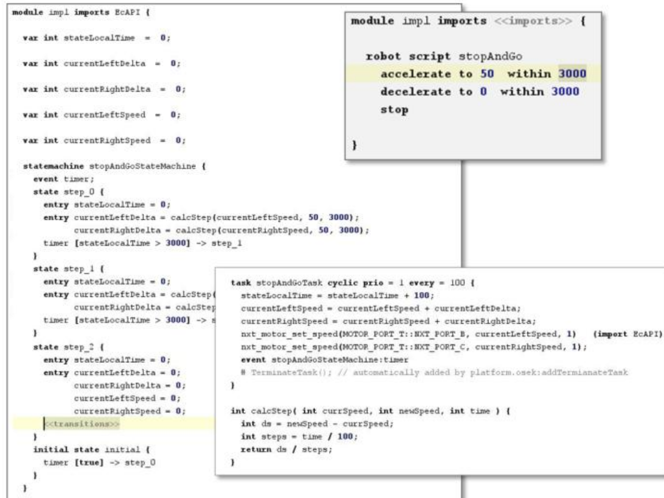

Figure 13. A simple robot script (top right, grey) and the lower level MEL program it is transformed into

5. Future Work

Future work will progress in three main directions: additional language concepts, real world-validation and integration of existing feature modeling tools.

Feature Modeling Tools

Currently we use a flat list of features (each basically boolean switches) as our feature model. We chose this approach because it is trivial to implement, and the point we wanted to make with our work was not to implement a new feature modeling tool.

However, to make our approach more useful in practice, integration with tools such as pure-variants [12]. We will simply import the list of features as well as the constraints among them. This will allow us to refer to these features from within feature expressions, and it will allow us to exploit the constraints between the features when we calculate whether a program is structurally valid.

Additional Language Concepts

An alternative to overlaying configuration over program or model code is to make the feature model the main configuration tool and add DSL code to it. In most feature management tools (for example, pure::variants [12]) features can have parameters. For example, when selecting a *buffered* feature for a communication protocol product line, a *buffer size* parameter can be specified. Generalizing this approach leads to the following:

- each feature may define any number of parameters. These cannot just be simple types (*int*, *string*, *boolean*) but can include DSLs.
- When a feature is selected, a value for the parameter that complies to the parameter's type has to be supplied. For DSL-typed parameters, this means that a model that conforms to the DSL must be supplied.

Because projectional language workbenches can integrate models using any combination of DSLs, this approach is feasible. Figure 14 shows a very early prototype of this; a retry algorithm is used as the value for the *retry* parameter of the *polling* feature.

Optionally taking away program elements if they are not included in the variant is only one way of implementing variability. The approach is often called negative variability. The other alternative is to conditionally add to a minimal core - positive variability. The advantage is that the minimal core remains small, quite in contrast to negative variability where the overall program that includes all variants can grow quite large. Like in AOP [2], positive variability requires pointcuts to define where to add the additional program elements to the core. Future work will focus on positive variability MEL as a means of implementing variability.

```
mandatory communication
  xor async
    or polling [ timeout : int
                 retry : custom retry script
                                wait 10 ms
                                try again
                                give up     ]
    or callback
    or oneway
      xor bestEffort [ noOfTries : int ]
      xor reliable
  xor sync
    optional exception
```

Figure 14. A DSL snippet in a feature model

As mentioned above, the current approach to feature validation is brute force and does not at all scale. One aspect of our future work will address this issue. We've already started collaborating with a university who has experience in this regard.

Real-World validation

The other main avenue of future work is real world validation. We are currently in the process of starting up a project to do a real prototype - something more realistic than the Lego Mindstorms example we are currently building. The connection to real requirements management systems and to variant management tools will be a part of this prototype.

6. Related Work

The idea of using DSLs to describe variability in product lines is not new. Various authors have published about this [13,14,15] and the approach is used in practice. The approach described in this paper is different since the various DSLs can be mixed and integrated. Language composition for textual languages is not easily possible with non-projectional editors, although progress is being made, as exemplified by [18, 28].

Overlaying configuration variability over customization variability has been done before, too. The C preprocessor can be used to this effect using *#ifdefs*. The approach can also be used on models. For example, Krzysztof Czarnecki and his group have overlaid feature-based variability over UML diagrams [16].The approach described in this paper is different in that configurative variability can be overlaid over models and code in the same way - there is no difference between the two in the first place. Since the feature expressions are also a formal language, the expressions can be formally checked and interpreted. The ability to show the program/model code with feature clauses enabled or not, and to show the (and edit!) the model in a variant-specific way is also radically different from these tools. CIDE, a specific solution for C code is described in [19]. However, the approach described in this paper is different since it works for any language within MPS. Also, the approach described in this paper supports the combination of the annotation-based approach with language composition and DSLs. VML [17] is another tool (based on Eclipse EMF) that

can map configurative variability to arbitrary models. However, since source code (C, Java) is not represented with EMF in Eclipse, a special solution had to be created to "adapt" VML to source code.

Showing statically that every valid variant of the feature model will result in a structurally valid program has been done before by [27] for the case of UML models and OCL constraints. Also the tool developed by Czarnecki et al. [16] has static validation to make sure that every variant of the UML model is structurally correct. Another approach for the same problem is described as part of the AHEAD methodology in [20]. Verifying that only "correct" programs are synthesized by program synthesis is a popular resarch topic [25, 26, 27,31]. We will use the approaches described in these papers in our future work, since the focus of our work is not primarily on this kind of verification.

Krzysztof Czarnecki and his group are currently working on a very interesing language: Clafer [24], a combination of structural class modeling and feature modeling. Krzysztof and his group are planning to integrate Clafer into the MPS prototype. One area where the approach described in this paper is more flexible is that we can use arbitrary DSLs and syntax to describe structural variability, whereas Clafer is essentially limited to class (or meta-) modeling.

7. Evaluation & Conclusion

As we continue to build Mindstorms applications with our language, it turned out that it is useful to extend plain C with embedded-specific concepts. Programs can be read and analyzed more easily: the more the language constructs resemble the intent of the programmer, the more meaningful analyses can be.

It is feasible to package the various aspects into separate language modules and make incremental extension possible. It is also surprisingly little effort to build language extensions: developing the basic C implementation has taken us about 3 weeks. Adding the statemachine facilities has been done in one afternoon. Creating the *robot routing* DSL on top was a matter of 4 hours, including the mapping down to tasks and state machines. Consequently, the concept of building DSLs to express some aspect of a product line is absolutely feasible.

Adding variability annotations to program elements is not fundamentally new. *#ifdefs* in C programs can be used for a similar approach. However, the ability to reliably evaluate the expressions, show and edit the programs in variant-specific ways as well as the static validation of feature dependencies has proven useful even in our simple examples.

Acknowledgments

My thanks go to Konstantin Solomatov of JetBrains who tirelessly supported my efforts of learning MPS, as well as to Christoph Elsner for his feedback on this paper.

References

[1] Fowler, M., *Language Workbenches: The Killer-App for Domain Specific Languages?*, http://martinfowler.com/articles/languageWorkbench.html

[2] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J. *Aspect-Oriented Programming*. Proceed ings of ECOOP 1997, vol.1241. pp. 220–242.

[3] Markus Voelter, *Embedded Software Development with Projectional Language Workbenches*, Proc. of MODELS 2010,

[4] MISRA Group, *Misra-C*, http://www.misra-c2.com/

[5] Gotel, O., Finkelstein, A., *An Analysis of the Requirements Traceability Problem*, Proc. of First International Conference on Requirements Engineering, 1994, pages 94-101

[6] Software Engineering Institute, *Software Product Lines*, http://www.sei.cmu.edu/productlines/

[7] Lego SA, *Mindstorms*, http://mindstorms.lego.com

[8] Sourceforge.net, *nxtOSEK*, http://lejos-OSEK.sourceforge.net/ Osek

[9] IBM Corp, *Requisite Pro - a Requirements Management Tool*, http://www-01.ibm.com/software/awdtools/reqpro/

[10] IBM Corp, *Rational DOORS*, http://www-01.ibm.com/ software/awdtools/doors/productline/

[11] Kang, K.C. and Cohen, S.G. and Hess, J.A. and Novak, W.E. and Peterson, A.S., *Feature-oriented domain analysis (FODA) feasibili ty study*, Technical Report CMU/SEI-90-TR-021, SEI, Carnegie Mellon University, November 1990

[12] Pure Systems GmbH, *pure::variants*, http://www.pure-systems.com/pure_variants.49.0.html

[13] Batory, D., Johnson, C., MacDonald, B., von Heeder, D., *Achieving Extensibility through Product-Lines and Domain-Specific Lan guages: A Case Study*, LNCS, Volume 1844/2000

[14] Mernik, M., Heering, J., Sloane, A., *When and how to develop domain-specific languages*, ACM Computing Surveys (CSUR), Volume 37 , Issue 4

[15] Tolvanen, J., Kelly, S., *Defining Domain-Specific Modeling Languages to Automate Product Derivation: Collected Experiences*, Lecture Notes in Computer Science, Volume 3714/2005

[16] Czarnecki, K., Antkiewicz, M.. *Mapping Features to Models: A Template Approach Based on Superimposed Variants*. In Proceed ings of GPCE'05, 2005

[17] Loughran, N., Sanchez, P., Garcia, A., Fuentes, L., *Language Support for Managing Variability in Architectural Models*, Lecture Notes in Computer Science, Volume 4954/2008

[18] Bravenboer, M., Visser, E., *Designing Syntax Embeddings and Assimilations for Language Libraries*, ATEM'07 and http://swerl.tudelft.nl/bin/view/EelcoVisser

[19] Kästner, C., *CIDE: Virtual Separation of Concerns*, http://wwwiti.cs.uni-magdeburg.de/~ckaestne/

[20] Thaker, S., Batory, D., Kitchin D., Cook, W., *Safe Composition of Product Lines*, GPCE 2007, http://userweb.cs.utexas.edu/~wcook/ papers/gpce07/ThakerGPCE07.pdf

[21] JetBrains Inc, Meta Programming System (MPS), http://jetbrains.com/mps

[22] Voelter, M., Solomatov, K., *Language Modularization and Composition with Projectional Language Workbenches illustrated with MPS*, submitted to SLE 2010

[23] Wikipedia, View Model, http://en.wikipedia.org/wiki/View_model

[24] Krzyszztof Czarnecki, *Feature and Class Models in Clafer: Mixed, Specialized, and Coupled*, personal communication, now probably available at http://gsd.uwaterloo.ca/~kczarnec/

[25] Huang S., Zook D., Smaragdakis, Y., *Statically Safe Program Generation with SafeGen*, GPCE 2005

[26] Krishnamurthi S., Fisler K.,, Greenberg M.. *Verifying Aspect Advice Modularly*, ACM SIGSOFT 2004

[27] Czarnecki, K., Pietroszek, K.. *Verification of Feature-Based Model Templates Against Well-Formedness OCL Constraints*. GPCE 2006

[28] Bravenboer, M., Visser, E., *Parse Table Composition, Separate Compilation and Binary Extensibility of Grammars,* SLE'08 and http://swerl.tudelft.nl/bin/view/EelcoVisser

[29] Gamma, E., Helm, R., Johnson R., Vlissides, J., *Design Patterns,* Addison-Wesley, 1994

[30] Continental Automotive, *http://osek-vdx.org/*

[31] Kästner, C., Apel, S., Trujillo, S., Kuhlemann M., Batory, D., *Guaranteeing Syntactic Correctness for all Product Line Variants: A Language-Independent Approach*, TOOLS Europe 2009