Product Line Engineering using Domain-Specific Languages

Markus Voelter
Independent/itemis
Stuttgart, Germany
Email: voelter@acm.org

Eelco Visser
Delft University of Technology
Delft, The Netherlands
Email: visser@acm.org

***Abstract*—This paper investigates the application of domain-specific languages in product line engineering (PLE). We start by analyzing the limits of expressivity of feature models. Feature models correspond to context-free grammars without recursion, which prevents the expression of multiple instances and references. We then show how domain-specific languages (DSLs) can serve as a middle ground between feature modeling and programming. They can be used in cases where feature models are too limited, while keeping the separation between problem space and solution space provided by feature models. We then categorize useful combinations between configuration with feature model and construction with DSLs and provide an integration of DSLs into the conceptual framework of PLE. Finally we show how use of a consistent, unified formalism for models, code, and configuration can yield important benefits for managing variability and traceability. We illustrate the concepts with several examples from industrial case studies.**

I. Introduction

The goal of product line engineering (PLE) is to efficiently manage a range of products by factoring out commonalities such that definitions of products can be reduced to a specification of their variable aspects. One way of achieving this is the expression of product configurations on a higher level of abstraction than the actual implementation. An automated mapping transforms the configuration to the implementation. Traditionally this higher level of abstraction is realized with feature models [5] or similar configuration formalisms such as orthogonal variability models [17] or decision models [10]. A feature model defines the set of valid configurations for a product in a product line by capturing all variations points (i.e. features), as well as the constraints between them.

However, since feature models can only describe bounded configuration spaces, their expressivity is limited. This limitation can be reduced to the fact that the feature modeling formalism corresponds to context-free grammars without recursion. By using full context-free grammars, the expressivity of feature modeling can be extended to domain-specific configuration languages with unbounded configuration spaces, without the need to regress to programming in a general-purpose programming language.

In this paper, we investigate the application of domain-specific languages (DSLs) in product line engineering. We first analyze the difference in expressivity between feature models and DSLs, and discuss when to use which approach (Section II). For a given product line, the decision between feature models and DSLs often is not mutually exclusive. We categorize approaches for composing feature models and DSLs, for example, by using feature models to configure DSL programs (Section III). If we express the complete product definition based on the linguistic integration of feature models and DSLs, we can reap a number of additional benefits such as uniform traceability (Section IV). To illustrate the approach, we discuss three industrial case studies of DSLs used in product line engineering (Section V). Finally, to put the approach into perspective, we present a mapping of DSLs and their tools to the core concepts and processes of PLE (Section VI).

II. From Feature Models to DSLs

A feature model is a compact representation of the features of the products in a product line, as well as the constraints imposed on configurations. Feature models are an efficient formalism for *configuration*, i.e. for *selecting* a valid combination of features from the feature model. The set of products that can be defined by feature selection is fixed and finite: each valid combination of selected features constitutes a product. This means that all valid products have to be ”designed into” the feature model, encoded in the features and the constraints among them. Some typical examples of variation points that can be modeled with feature models are the following:

- Does the communication system support encryption?
- Should the in-car entertainment system support MP3s?
- Should the system be optimized for performance or memory footprint?
- Should messages be queued? What is the queue size?

In the simplest case, product lines can be implemented with programming languages only. For example, an object-oriented framework is implemented as part of domain engineering, and it is then customized specifically for each product by writing framework client code. The advantage of this code-only approach is that no special tools are needed and that there is a high degree of flexibility for the implementers. Any kind of variability can be expressed with programming languages, using techniques such as preprocessors, branching, polymorphism, generics or design patterns. However, using these techniques only, there is no distinction between the problem space and the solution space — no higher level representation of the variability in a product line is provided, making it hard to keep track of, and manage the variability. This is especially problematic if the definition of a product requires consistent changes in several places in the implementation artifacts.

Feature models provide an abstraction over the implementation of the product. Based on the feature configuration, mappings derive the necessary adaptations of the underlying implementation artifacts. Because of this abstraction, feature model-based configuration is simple to use — product definition is basically a decision tree. This makes product configuration efficient, and potentially accessible for stakeholders other than software developers.

As described by Batory [4] and Czarnecki [9], a particular advantage of feature models is that a mapping to logic exists. Using SAT solvers, it is possible to check, for example, whether a feature model has valid configurations at all. The technique can also be used to automatically complete partial configurations. This has been shown to work for realistically-sized feature models [15]. Pure::variants (http://pure-systems.com) maps feature models to Prolog to achieve a similar goal, as does the GEMS-based tool described in [23].

In the rest of this section we will discuss the limitations of feature models. We argue that in cases in which feature models are unsuitable, we should not regress to low-level programming, but use DSLs instead, to avoid losing the differentiation between problem space and solution space. As an example we use a product line of water fountains as found in recreational parks[1]. Fountains can have several basins, pumps and nozzles. Software is used to program the behavior of the pumps and valves to make the sprinkling waters aesthetically pleasing. The feature model in Fig. 1 represents valid hardware combinations for a simple water fountain product line. The features correspond to the presence of a hardware component in a particular fountain installation.

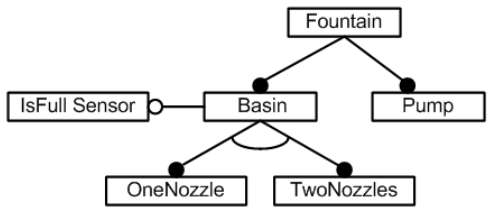


Fig. 1. Feature model for the simple fountains product line used as the example. Fountains have basins, with one or two nozzles, and an optional full sensor. In addition, fountains have a pump.

The real selling point of water fountains is their *behavior*. A fountain's behaviour determines how much water each pump should pump, at which time, with what power, or how a pump reacts when a certain condition is met, e.g. a basin is full. Expressing the full range of such behaviors is not possible with feature models. Feature models can be used to select among a fixed number of predefined behaviors, but approximating all possible behaviors would lead to unwieldy feature models. Instead we could program the behavior in a general purpose language such as C. However, this means that we lose the abstraction feature models provide, reducing efficiency and understandability, as well as the possibility for direct involvement of non-programmers. Domain-specific languages can serve as a middle ground between the controlled setting of feature model-based configuration and the complete lack of restrictions of general purpose programming languages.

A. Feature Models as Grammars

To understand the limitations of feature models, consider their relation to grammars. Feature models essentially correspond to context-free grammars without recursion [8]. For example, the feature model in Fig. 1 is equivalent to the following grammar. We use all caps to represent terminals, and camel-case identifiers as non-terminals:

```
Fountain -> Basin PUMP
Basin -> ISFULLSENSOR? (ONENOZZLE | TWONOZZLES)
```

This grammar generates a finite number of sentences, i.e there are exactly four possible configurations, which correspond to the finite number of products in the product line. However, this formalism does not make sense for modeling behavior, for which there is typically an infinite range of variability. To accommodate for unbounded variability, the formalism needs to be extended. Allowing recursive grammar productions is sufficient to model unbounded configuration spaces, but for convenience, we consider also attributes and references.

Attributes express properties of features. For example, the *PUMP* could have an integer attribute *rpm*, representing the power setting of the pump. Some feature modeling tools (e.g. pure::variants) support attributes.

```
Fountain -> Basin PUMP(rpm:int)
Basin -> ISFULLSENSOR? (ONENOZZLE | TWONOZZLES)
```

Recursive grammars can be used to model repetition and nesting. Repetition is also supported by cardinality-based feature models, as described in [8]. Nesting is necessary to model tree structures such as those occurring in expressions. The following grammar extends the fountain feature model with a *Behavior*, which consists of a number of *Rules*. The *Basin* can now have any number of *Nozzles*.

```
Fountain -> Basin PUMP(rpm:int) Behavior
Basin -> ISFULLSENSOR? NOZZLE*
Behavior -> Rule*
Rule -> CONDITION CONSEQUENCE
```

References allow the creation of context-sensitive relations between parts of generated sentences — or subtrees of the generated derivation trees. For example, by further extending our fountain grammar we can describe a rule whose condition refers to the *full* attribute of the *ISFULLSENSOR* and whose consequence sets a *PUMP*'s *rpm* to 0.

```
Fountain -> Basin id:PUMP(rpm:int)? Behavior
Basin -> id:ISFULLSENSOR(full:boolean)? id:NOZZLE*
Behavior -> Rule*
```

[1]This is an anonymized version of an actual project the authors have been working on. The real domain was different, but the example languages presented in this paper have been developed and used for that other domain.

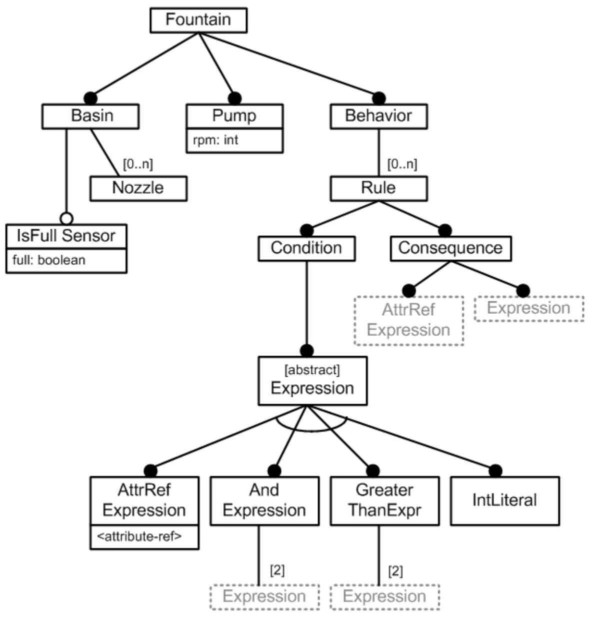


Fig. 2. An extended feature modeling formalism is used to represent the example feature model with attributes, recursion and and references (the dotted boxes).

```
Rule -> Condition Consequence
Condition -> Expression
Expression -> ATTRREFEXPRESSION | AndExpression |
             GreaterThanExpression | INTLITERAL;

AndExpression -> Expression Expression
GreaterThanExpression -> Expression Expression

Consequence -> ATTRREFEXPRESSION Expression
```

Fig. 2 shows a possible rendering of the grammar with an enhanced feature modeling notation. We use cardinalities, as well as references to existing features, the latter are shown as dotted boxes. A valid configuration could be the one shown in Fig. 3. It shows a fountain with one basin, two nozzles named *n1* and *n2*, one sensor *s* and a pump *p*. It contains a rule that expresses that if the *full* attribute of *s* is set, and the *rpm* of pump *p* is greater than zero, then the *rpm* should be set to zero.

B. Domain-Specific Languages

While the extended grammar formalism discussed above enables us to cover the full range of behavior variability, the use of trees to instantiate these grammars is not practical. Another interpretation of these grammars is as definition of a *language* with a *textual concrete syntax* — the tree in Fig. 3 looks like an abstract tree (AST). To make the language readable we need to add concrete syntax definitions (keywords), as in the following extension of the fountain grammar:

```
Fountain -> "fountain" Basin Pump Behavior
Basin -> "basin" IsFullSensor Nozzle*
Behavior -> Rule*

Rule -> "if" Condition "then" Consequence
```

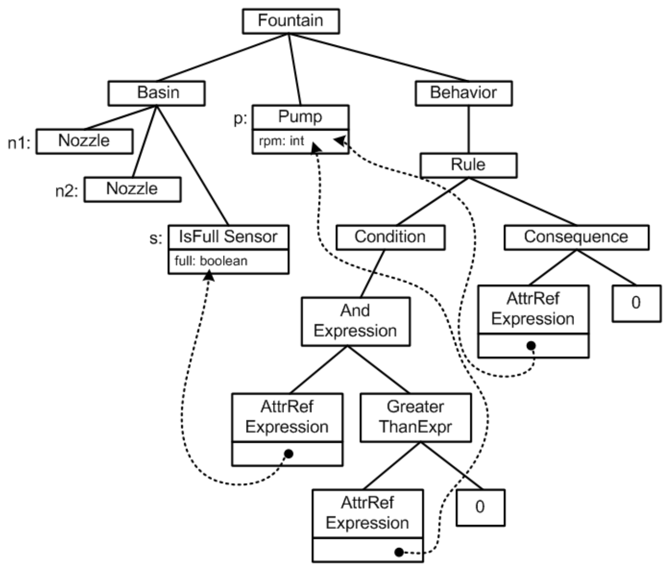


Fig. 3. Example configuration using a tree notation. Referenceable identities are rendered as labels left of the box. The dotted lines represent references to variables.

```
Condition -> Expression
Expression -> AttrRefExpression | AndExpression |
             GreaterThanExpression | IntLiteral;

AndExpression -> Expression "&&" Expression
GreaterThanExpression -> Expression ">" Expression
AttrRefExpression -> <attribute-ref-by-name>
IntLiteral -> (0..9)*

Consequence -> AttrRefExpression "=" Expression

IsFullSensor: "sensor" ID (full:boolean)?
Nozzle: "nozzle" ID
Pump: "pump" ID (rpm:int)?
```

We can now write a program that uses a convenient textual notation, which is especially useful for the expressions in the rules. We have created a *domain-specific language* for configuring the composition *and* behavior of fountains. A complete language definition would also include typing rules and other constraints, but that is beyond the scope of this paper.

```
fountain
  basin sensor s
        nozzle n1
        nozzle n2
  pump p
  if s.full && p.rpm > 0 then p.rpm = 0
```

DSLs fill the gap between feature models and programming languages. They can be more expressive than feature models, but they are not as unrestricted and low-level as programming languages. Like programming languages, DSLs, support *construction*, allowing the composition of an unlimited number of programs. Construction happens by instantiating language concepts, establishing relationships, and defining values for attributes. We do not a-priori know all possible valid programs. In contrast to programming languages, DSLs keep the distinction between problem space and solution space intact since they consist of concepts and notations relevant to the problem domain. Non-programmers can continue to contribute directly to the product development process, without being exposed to implementation details.

DSLs are a good fit when instances of concepts need to be created, when relationships between these instances must be established, or when algorithmic behavior has to be described, e.g. in business rules, calculations, or events. Examples of the application of DSLs include calculating the VAT and other taxes in an invoicing product line, specifying families of pension contracts, and defining communication protocols in embedded systems.

In general, a *domain-specific language* (DSL) is a software language specialized for a particular problem domain. DSLs can use graphical, textual or tabular concrete syntax, or any combination thereof. Like programming languages, DSLs can either be compiled — typically through code generation to a programming language — or interpreted by an interpreter running on the target environment.

A DSL's concrete and abstract syntax are tailored closely to the domain at hand. Using DSLs only requires knowledge about the problem domain, not about the solution domain. This improves productivity, quality and maintainability. Productivity is improved because a higher level notation is provided, avoiding dealing with implementation details. Quality is improved because transformations or interpreters execute the programs consistently. Maintainability is improved because changes to the program can be done on the level of the DSL program, or by changing the generator or interpreter.

A note on terminology: we use the terms *DSL program*, *DSL code* and *model* interchangably. Strictly speaking, the DSL code is a textual representation of a model, but this distinction is not relevant in this paper. While we focus primarily on textual DSLs in this paper, the discussion is equally valid for DSLs using other notations, as long as the underlying expressivity is not reduced, as for example in purely tabular notations. However, expression-like models can best be expressed using a textual notation.

Supplying convenient editors and other tools along with the DSL increases the usability of the language significantly. The term *language workbench* has been introduced by Martin Fowler in 2004 [11], referring to tools that support the efficient definition, composition and use of DSLs. Open Source examples include Eclipse Xtext (http://eclipse.org/xtext), Spoofax (http://strategoxt.org/Spoofax) and JetBrains MPS (http://www.jetbrains.com/mps/).

III. Combining DSLs and Feature Models

In the previous sections we have explained the difference in expressive power between feature models and DSLs. We have also outlined the benefits and drawbacks of both approaches. In this section we categorize how both approaches can be combined.

A. Implementing Components with DSLs

One way of using feature models is to use them to select between a number of prebuilt, reusable components that are used to customize a framework as part of product definition. Instead of implementing these components in a programming language, they can be implemented using DSLs. This is especially useful if the behavior in these components is highly domain-specific. At the time of product definition using the feature model, the domain expert does not have to know that the components have been developed using DSLs .

For example, we have worked on a system in which a DSL was used to describe OSGi component structures, generating all the low level OSGi details. Feature model-based configuration was then used to create different products from these components.

B. Variation over models

Feature models can also be used to vary the model in a fine-grained way. Model execution then happens in two steps: first, the model is configured based on the feature configuration, and then the configured model is processed as before via transformation, generation or interpretation.

Similar to configuration of source code, for example via the C preprocessor, the configuration of models can be done in several different ways. The actual implementation may be different depending on the DSL and language workbench:

- *Negative Variability via Removal* DSL program elements can be annotated with *presence conditions*, Boolean expressions over the features of a feature model. When the DSL program is mapped to the solution space, a model transformation removes all those elements whose presence condition is false based on the current feature configuration. Czarnecki and Antkiewicz show this approach applied to UML models [6], including static checks that ensure that every valid feature configuration leads to a structurally correct UML model.
- *Positive Variability via Superimposition* A set of prebuilt model fragments is created. The feature configuration selects a subset of them. The fragments are then merged using some DSL-specific or generic merge operator, resulting in a superimposed model representing the variant. Apel et al discuss the approach in general and demonstrate it for various UML diagrams, among them class diagram, state diagrams and sequence diagrams. [1].
- *Positive Variability via Aspects* A core program is available, together with a set of aspects. The aspects use pointcuts to define where and how they affect the core program. Based on the feature configuration, a subset of these aspects is selected and applied to the core program. The case study in [22] describes this approach applied to EMF models (Eclipse Modeling Framework, http://eclipse.org/emf).

As Fig. 4 shows, this approach reduces the numbers of variation points in the artifacts and automatically supports the consistent implementation of several, non-local adaptations. Since the models are transformed into lower-level artifacts automatically, they "expand" the variability to potentially many low-level variation points.

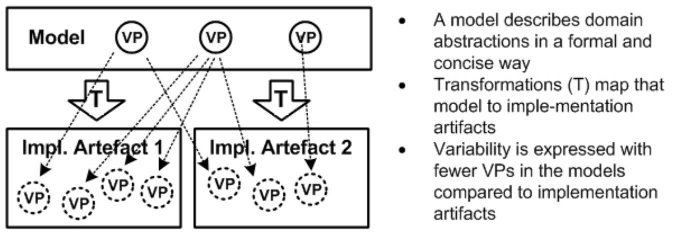

Model
VP
T
Impl. Artefact 1
Impl. Artefact 2
A model describes domain abstractions in a formal and concise way
Transformations (T) map that model to imple-mentation artifacts
Variability is expressed with fewer VPs in the models compared to implementation artifacts

Fig. 4. Transformations "expand" variation points, thereby reducing the number of variation points that have to be management explicitly.

C. Variations in the Transformation or Execution

When working with DSLs, the execution of models — by transformation, code generation or interpretation — is under the control of the domain engineer. The transformations or the interpreter can also be varied based on a feature model.

- *Negative Variability via Removal* The transformations or the interpreter can be annotated with presence conditions, the configuration happens before the transformations or the interpreter are executed.
- *Branching* The interpreter or the transformations can query over a feature configuration and then branch accordingly at runtime.
- *Positive Variability via Superimposition* Transformations or interpreters can be composed via superposition before execution. For transformations, this is especially feasible if they transformation language is declarative, which means that the order in which the transformations are specified is irrelevant. Interpreters are usually procedural, object-oriented or functional programs, so declarativeness is hard to achieve in those.
- *Positive Variability via Aspects* If the transformation language or the interpreter implementation language support aspect oriented programming, then this can be used to configure the execution environment. For example, the Xpand code generation engine (http://wiki.eclipse.org/Xpand) supports AOP for code generation templates.

Examples for all of these are described by Voelter and Groher [19] based on the openArchitectureWare tool suite[2].

Creating transformations with the help of other transformations or by any of the above variability mechanisms is also referred to as *higher-order transformations* [16]. Note that if a bootstrapped environment is used, the transformations are themselves models created with a transformation DSL. This case then reduces to just variation over models, as described in the previous subsection.

D. DSLs as Attribute Types

Some feature modeling tools support feature attributes. Typically, the types of these attributes are primitive (integer, string, float). They could also be typed with a DSL, the set of valid programs expressed in this DSL would be the range of values.

This approach is useful when the primary product definition can be expressed with a feature model. The DSL-typed attributes can be used for those variation points for which selection is not expressive enough.

E. Feature Models on Language Elements

The opposite approach is also possible. The primary product definition is done with DSLs. However, some language elements may have a feature model associated with them for detailed configuration. When the particular language concept is instantiated, a new ("empty") feature configuration is created, and can be configured by the application engineer.

F. Merging of the two approaches

We have described the limitations of the feature modeling approach. The feature modeling community is working on alleviating some of these limitations.

For example, cardinality based feature models [8] support the multiple instantiation of feature subtrees. References between features could be established by using feature attributes typed with another feature — the value range would be the set of instances of this feature. Name references are an approximation of this approach.

Clafer [2] combines meta modeling and feature modeling. In addition to providing a unified syntax and a semantics based on sets, Clafer also provides a mapping to SAT solvers to support validation of models. The following code is Clafer code (adapted from Michal Antkiewicz' *Concept Modeling Using Clafer* tutorial at http://gsd.uwaterloo.ca/node/310).

```
abstract Person
  name : String
  firstname : String
  or Gender
     Male
     Female
  xor MaritalStatus
    Single
    Married
    Divorced
  Address
    Street : String
    City : String
    Country : String
    PostalCode : String
    State : String ?

abstract WaitingLine
  participants -> Person *
```

The Clafer code example describes a concept *Person* with the following characteristics:

- a name and a first name of type *String* (similar to attributes)
- a gender which is *Male* or *Female*, or both (similar to or-groups in feature models)
- a marital status which is either *single*, *married* or *divorced* (similar to xor-groups in feature models)
- an *Address* (similar composition is language definitions)

[2]openArchitectureWare has since been migrated to the Eclipse Xpand and Xtext projects

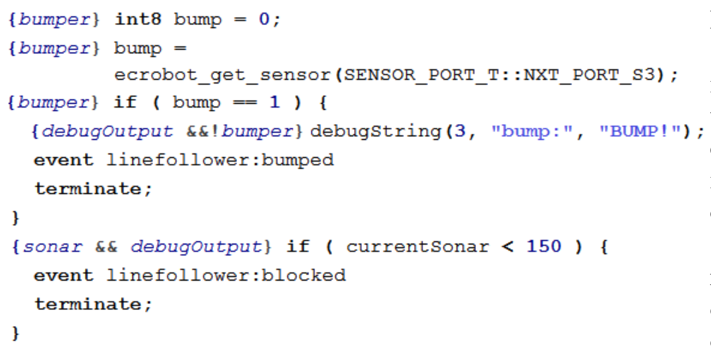
- and an optional *State* attribute on the address (similar to optional features in feature modeling)

The code also shows a reference: a *WaitingLine* refers to any number of *Persons*.

Note, however, that an important ingredient for making DSLs work in practice is the domain-specific concrete syntax. None of the approaches mentioned in this section provide customizable syntax. However, approaches like Clafer are a very interesting backend for DSLs to support analysis, validation and automatic creation of valid programs from partial configurations (Section VII).

IV. Benefits of Tools

If all product line artifacts, i.e. program code, models, and transformations, are expressed using a single modeling infrastructure, then only a single approach is needed for varying any of them. In addition, the integration between different models representing different aspects of the overall product configuration becomes simpler. For example, the ability to refer to features in a feature model from an Xtext-based language is a generic, reusable module that can be integrated into arbitrary DSLs. Similar tooling is available for JetBrains MPS. Arbitrary program elements expressed with any language defined in MPS can be annotated with presence conditions (Fig. 5). Upon transformation, all those model elements whose presence condition is false at the time of generation are removed, so no lower level code is generated from them. It is also possible to configure the program for a specific variant while it is edited.



```
{bumper} int8 bump = 0;
{bumper} bump =
         ecrobot_get_sensor(SENSOR_PORT_T::NXT_PORT_S3);
{bumper} if ( bump == 1 ) {
  {debugOutput &&!bumper} debugString(3, "bump:", "BUMP!");
  event linefollower:bumped
  terminate;
}
{sonar && debugOutput} if ( currentSonar < 150 ) {
  event linefollower:blocked
  terminate;
}
```

Fig. 5. Presence conditions (Boolean expressions over features) on program elements in the MPS tool. Presence conditions can be attached to any program node and control which nodes are transformed during code generation.

A similar approach can be used for expressing traceability, another important cornerstone of PLE [13]. Arbitrary model elements can be annotated with traceability links to a requirements database. For Eclipse, the VERDE project (http://www.itea-verde.org/) develops generic traceability tooling with which any EMF-based model can be related to other models or RIF-based requirements files. In MPS this is possible as well, using the same annotation-based approach that is used for presence conditions. MPS' capability to combine independently developed languages makes the composition of an overall product configuration from program/model fragments expressed in various languages almost trivial. Language composition in MPS and the annotation mechanism is explained in detail in [20].

V. Examples

This section contains industry examples in which DSLs are used to implement product lines. Note that we cannot reveal the actual companies using the DSL, and in case of the fountains, which we had already introduced in Section II, we even had to move the example into another domain. However the cases and the languages are real world examples.

A. Alarm System Menus

The company manufactures burglar alarm systems. These systems detect when buildings are compromised. They consist of sensors that detect the burglary, and actuators including sirens, lights, and alarm propagation facilities to the police. These systems also have configuration devices, used by the house owner to configure, among other things, when the system should be active, and which kinds of alarms should be raised under which conditions.

The company sells many different alarm systems, sensors, and actuators. Based on the configuration of an individual system, the menus in the configuration device need to be adapted. Traditionally these menus have been described using Word documents, developers then implemented the menus in C.

A new approach uses a DSL that formally describes the menu structure. Code generation creates the C implementation. Fig. 6 shows some example code. The language is purely structural, no behavior is described explicitly. A templating mechanism is provided that supports multiple instantiations of the same template in several locations in the tree, configuring each instance with different values passed into the template instance. Menus can also inherit from other menus to avoid code duplication for related systems.

While the DSL is relatively simple, it plays an important role nonetheless, because product management can directly describe the menus in a formal way, making the overall development process much simpler, faster, and less error prone.

We have used a DSL instead of a feature model for the following reasons: menus require recursion in the underlying formalism to be able to define unlimited numbers of instances of submenus. The ability to define standalone submenus that can be included in other menus is an example of references. Finally, the various elements have many fine grained attributes. A textual notation works much better in these cases than trees.

B. Fountains

This is the example used in section II. Before using DSLs, fountain designers experimented with different arrangements of basins and pumps, writing down, in prose text, interesting configurations and behaviors. Developers wrote the corresponding controller code in C. The domain from which this

```
menu Normal label "Standardmenue"
    submenu autoLocking label "Automatic Locking"
        item startTime sys(TurnOnAlarm)
            valuerange Time
        item endTime sys(TurnOffAlarm)
            valuerange Time
        template areaSettings [size=15, area=1,
                 sw=sys(TurnOnAlarm)] area1Settings
        template areaSettings [size=10, area=2,
                 sw=sys(TurnOnAlarm)] area2Settings

    end
end

template [size: int, area: int, sw: swref] areaSettings
    item onOrOff sys(TurnOffAlarm) labelexpr "Autolock "
                                       +area+" on/off"
        bool true = label(size) "On" false = label "Off"
    item test sys(AlarmLevel) label "Test"
        bool
    item alarmLevel sys(AlarmLevel)
        valuerange SoundLevel restrict size..80
end

menu Expert extends Normal
    item master sys(UnlockNow) afterItem unlockNow bool
end
```

Fig. 6. Example menu definition for the alarm systems. Menus contain submenus and items. Templates can be defined, and template instances can be embedded into a menu tree several times with different parameter values.

example is derived is very complex, with ca. 700 different products!

Several DSLs are used. The first one (Fig. 7) is used to describe the logical structure of basins, pumps and valves. It uses a form of multiple inheritance similar to classes (here: appliances) and traits (here: features), quite similar to what Scala provides (http://scala-lang.org).

```
feature BasicOnePump
    pump compartment cc1
        static compressor c1

feature AtLeastOneZone extends BasicOnePump
    water compartment comp1
        pumped by c1
        compartment levelsensor ct_f1
        light l_f1

feature[f] SuperPowerCompartment
    water compartment adds to f
        superPowerMode

feature WithAlarm
    level alarm a1

fountain StdFountain extends AtLeastOneZone
```

Fig. 7. Fountains can be composed from features, who contribute hardware elements. Parametrized features (those with brackets) can be included more than once, binding the parameter differently each time.

A second language (Fig. 8) is used to describe the behavior. The behavior model refers to a hardware structure. All the hardware elements have events, properties and commands defined, which can be accessed from the behavior model. A state based, reactive, asynchronous language is used to describe the behavior, driving the activation of the pumps and valves based on sensor input and timing events.

In addition to generating C code, there is also an in-IDE interpreter that can run the pumping programs and execute

```
pumping program P1 for AtLeastOneZone + WithAlarm +
                       SuperPowerCompartment[f=comp1]  {

    parameter defaultWaterLevel : int
    parameter superWaterLevel: int
    event superPowerTimeout

    init {
        set comp1->targetHeight = defaultWaterLevel
    }

    start:
        on (comp1->needsPower == true) && !(comp1->isPumping) {
            do comp1->pumpOn
        }
        on comp1->enough {
            do comp1->pumpOff
        }
        on comp1.superPumping->turnedOn {
            set comp1->targetHeight = superWaterLevel
            raise event superPowerTimeout after 20
        }
        on comp1.superPumping->turnedOff or superPowerTimeout {
            set comp1->targetHeight = defaultWaterLevel
        }
}
```

Fig. 8. The fountain behavior is defined with a reactive, asynchronous language. A pumping program is defined for a combination of hardware features. The properties, events and commands of the hardware elements defined by these features can be used in the programs.

tests. These tests are also described with a textual language. A simulation engine to ”play” with the programs is available as well. This is an example where additional tools make the use of DSLs much more feasible for domain experts.

The behavior language also overlays configuration over the pumping behavior, an example of negative variability of a model (Section III). The behavior can be varied depending on whether certain optional hardware components are installed in the fountain, as shown in Fig. 9.

We have used DSLs in this case because algorithmic behavior is described. The expressions used in the language cannot sensibly be represented with feature models. The hardware structure language has to be able to instantiate the same hardware component several times, another reason for using a DSL instead of a feature model. We use negative variability to overlay hardware structure dependencies over the behavior specifications.

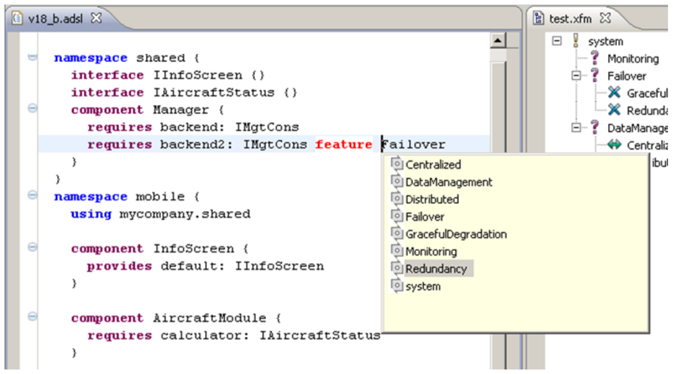
```
every 10 {
    variant WithAlarm {
        if ( comp1->currentHeight > alarmLevel ) {
            do a1->ring
            set alarmActive = true
        }
        if alarmActive {
            if comp1->currentHeight > alarmLevel {
                do a1->stop
                set alarmActive = false
            }
}   }   }
```

Fig. 9. Within a pumping program, *variant* statements can be used to implement negative variability over optional hardware features. The example shows code that is only executed if the *WithAlarm* feature is present.

C. Architecture DSLs

Many product lines are built on a common software architecture, while the application functionality varies from product to product. Architecture DSLs [21], [12] can be used very effectively in these cases. An architecture DSL is a DSL, in which the abstractions of the language correspond to the architectural concepts of the execution platform. They are defined by architects, and used by developers as they develop applications. When developing products in challenging environments such as distributed real-time embedded systems, architecture DSLs can provide the benefit of simulation and automatic optimization as described by Balasubramanian and Schmidt in [3].

One project in the transportation industry has used an architecture DSL overlaid with configuration-based variability. The architecture models directly refer to the features defined in the configuration model, a form of presence conditions. Tooling is based on Eclipse Xtext and pure::variants. Fig. 10 shows a screenshot. The system also supported AOP: aspects are available to "contribute" additional properties to existing model elements. The article in [21] describes this example, and the general approach, in more detail.



```
namespace shared {
  interface IInfoScreen {}
  interface IAircraftStatus {}
  component Manager {
    requires backend: IMgtCons
    requires backend2: IMgtCons feature Failover
  }
}
namespace mobile {
  using mycompany.shared

  component InfoScreen {
    provides default: IInfoScreen
  }

  component AircraftModule {
    requires calculator: IAircraftStatus
  }
```

Fig. 10. Tool support (Eclipse Xtext and pure::variants) for referring to feature models from DSLs, implementing negative variability over DSL code.

A DSL was used in this case because architecture definition makes heavy use of identities and references, attributes and recursion. For example, it must be possible to define any number of components, and these than have to be instantiated and connected. So even while no expressions are used, DSLs are still useful in this case.

VI. Conceptual Mapping from PLE to DSLs

This section looks at the bigger picture of the relationship between PLE and DSLs. It contains a systematic mapping from the core concepts of PLE to the technical space of DSLs. First we briefly recap the core PLE concepts.

- *Core Assets* designate reusable artifacts that are used in more than one product. As a consequence of their strategic relevance, they are usually high quality and maintained over time. Some of the core assets might have variation points.
- *A Variation Point* is a well-defined location in a core asset where products differ from one another.
- *Kind of Variability* classifies the degrees of freedom one has when binding the variation point. This ranges from setting a simple Boolean flag over specifying a database URL to a DSL program to a Java class hooked into a platform framework.
- *Binding Time* denotes the point in time when the decision is made as to which alternative should be used for a variation point. Typical binding times include source time (changes to the source code are required), load time (bound when the system starts up) and runtime (the decision is made while the program is running).
- *The Platform* are those core assets that actually form a part of the running system. Examples include libraries, frameworks or middleware.
- *Production Tools* are core assets that are not part of the platform, but are used during the possibly automated development of products.
- *Domain Engineering* refers to activities in which the core assets are created. An important part of domain engineering is domain analysis, during which a fundamental understanding of the domain and its commonalities and variability is established.
- *Application Engineering* is the phase in which the domain engineering artifacts are used to create products. Unless variation points use runtime binding, they are bound during this phase.
- *The Problem Space* refers to the application domain in which the product line resides. The concepts found in the problem space are typically meaningful to non-programmers as well.
- *The Solution Space* refers to the technical space that is used to implement the products. In case of *software* product line engineering, this space is software development. The platform lives in the solution space. The production tools create or adapt artifacts in the solution space based on a specification of a product in the problem space.

In the following sections we now elaborate on how these concepts are realized when DSLs are used.

A. Variation Points and Kinds of Variability

This represents the core of the paper and has been discussed extensively above: DSLs provide more expressivity than feature models, while not being completely unrestricted as programming languages.

B. Domain Engineering and Application Engineering

As we develop an understanding of the domain, we classify the variability. If the variability at a particular variation point is suitable for DSLs, we develop the actual languages together with the IDEs during domain engineering. The abstract syntax of the DSL constitutes a formal model of the variability found at the particular variation point. This is similar to analysis models, with the advantage that DSLs are executable. Users can immediately express exemplary domain structures or behavior and thereby validate the DSL. This should be exploited: language definition should proceed incrementally and iteratively, with user validation after each iteration. The example models created in this way should be kept around, they constitute unit tests for the language.

The combination of several DSLs is often necessary. Different variation points may have different DSLs that must be used together to describe a complete product. In the fountains example, one DSL describes the hardware structure of the fountains, and another one describes the behavior. The behavior DSL refers to elements from the hardware DSL for sensor values and events. In this case, one language merely refers to an element of another language. Deeper integration may also be necessary. For example, we may want to embed a reusable expression language into the fountain behavior DSL. Different language workbenches support language composition to different degrees. References between languages are always possible. Language embedding is not yet mainstream. It supported for example by Spoofax [14] and MPS [20].

Application engineering involves using the DSLs to bind the respective variation points. The language definition, the constraints, and the IDE guide the user along the degrees of freedom supported by the DSL.

C. Problem Space and Solution Space

DSLs can represent any domain. They can be technical, inspired by a library, framework or middleware, expected to be used by programmers and architects. DSLs can also cover application domains, inspired by the application logic for which the application is built. In this case they are expected to be used by application domain experts. In the case of application DSLs, the DSL resides in the problem space. For execution they are mapped to the solution space by the production tools. Technical DSL can, however, also be part of the solution space. In this case, DSL programs are possibly created by the mapping of an application domain DSL to the solution space. This is an example of cascading [18], [7]: one DSL is executed by mapping it to another DSL. It is also possible that technical DSLs are used by developers as an annotation for the application domain DSLs, controlling the mapping to the solution space or configuring some technical aspect of the solution directly [18].

D. Binding Time

DSLs can be executed in two ways. *Transformation* maps a DSL program to another formalism for which an execution infrastructure already exists. If this formalism is another DSL, we speak of model-to-model transformation, or simply transformation. If the target is a programming language, we speak of code generation. Alternatively, the DSL can also be interpreted: a meta program that is part of the platform executes the DSL program directly.

- If we generate source code that has to be compiled, packaged and deployed, the binding time is source. We speak of static variability, or static binding.
- If the DSL programs are interpreted, and the DSL programs can be changed as the system runs, this constitutes runtime binding, and we speak of dynamic variability.
- If we transform the DSL program into another formalism that is then interpreted by the running system, we are in a middle ground. It depends on the details of how and when the result of the transformation is (re-)loaded into the running system whether the variability is load-time or runtime.

When to use transformation vs. interpretation depends on various, usually non-functional concerns. Transformation and code generation is the more mainstream approach because most people find it easier to implement and debug. It has a couple of advantages compared to interpretation, better performance being the most important one, especially in embedded systems. Interpretation is intriguing because of the fast turnaround time. A detailed discussion of the trade-offs is beyond the scope of this paper.

E. Core Assets, Platform and the Production Tools

DSLs constitute core assets, they are used for many, and often all of the products in the product line. It is not so easy to answer the question whether they are part of the platform or the production tools:

- If the DSL programs are transformed, the transformation code is a production tool. It is used in the production of the products. The DSL or the models are not part of the running system.
- In case of interpretation, the interpreter is part of the platform. Since it directly works with the DSL program, the language definition becomes a part of the platform as well.
- If we can change the DSL programs as the system runs, even the IDE for the DSL is part of the platform.
- If the DSL programs are transformed into another formalism that is in turn interpreted by the platform, then the transformations constitute production tools and the interpreter of the target formalism is a part of the platform.

VII. Future Work

Hybrid solutions such as Clafer and their relationships to DSLs require further investigation. One question is whether Clafer is suitable as a verification backend for DSLs: a DSL's abstract syntax, including possible selection-based parts, could be translated into Clafer, and then Clafer's integration with solvers could be used to verify and check DSL programs. This approach promises a simplification over directly integrating DSLs with solvers, because Clafer provides abstractions that are much closer to DSLs than raw logic. A second, related question is which kinds of languages are suitable for this kind of integration. A third question relating to Clafer is whether it could be directly used as the abstract syntax formalism for DSLs. In this case, Clafer models would be annotated with concrete syntax definitions to render a textual DSL.

VIII. Conclusion

In this paper we have positioned domain-specific languages into the context of PLE. Our practical experience shows that DSLs play an important role in PLE, filling the expressive gap between feature models and programming languages. DSLs can provide problem space-level, formalized descriptions of the core application logic, that is hard to capture with feature models. We show that DSLs fit well with the existing feature model-based PLE approaches and the overall PLE approach. Tool support for DSLs, and for the integration between DSLs and feature models this is coming along. Finally, it is worth pointing out that language workbenches are becoming more and more powerful and user friendly, making the development of DSLs and their tools much less effort than language construction has historically been. The language workbench competition webpage at http://www.languageworkbenches.net/ provides a good overview.

References

[1] S. Apel, F. Janda, S. Trujillo, and C. Kästner. Model superimposition in software product lines. In *Proceedings of the Second International Conference on Model Transformation (ICMT)*, volume 5563 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, JUN 2009.

[2] K. Bak, K. Czarnecki, and A. Wasowski. Feature and meta-models in clafer: Mixed, specialized, and coupled. In *3rd International Conference on Software Language Engineering*, Eindhoven, The Netherlands, 10/2010 2010.

[3] K. Balasubramanian and D. C. Schmidt. Physical assembly mapper: A model-driven optimization tool for qos-enabled component middleware. In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2008, April 22-24, 2008, St. Louis, Missouri, USA*. IEEE Computer Society, 2008.

[4] D. S. Batory. Feature models, grammars, and propositional formulas. In J. H. Obbink and K. Pohl, editors, *Software Product Lines, 9th International Conference, SPLC 2005, Rennes, France, September 26-29, 2005, Proceedings*, volume 3714 of *Lecture Notes in Computer Science*. Springer, 2005.

[5] D. Beuche, H. Papajewski, and W. Schröder-Preikschat. Variability management with feature models. *Science of Computer Programming*, 53(3), 2004.

[6] K. Czarnecki and M. Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In R. Glück and M. R. Lowry, editors, *Generative Programming and Component Engineering, 4th International Conference, GPCE 2005*, volume 3676 of *Lecture Notes in Computer Science*, Tallinn, Estonia, 2005. Springer.

[7] K. Czarnecki, M. Antkiewicz, and C. H. P. Kim. Multi-level customization in application engineering. *Communications of the ACM*, 49(12), 2006.

[8] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1), 2005.

[9] K. Czarnecki and A. Wasowski. Feature diagrams and logics: There and back again. In *Software Product Lines, 11th International Conference, SPLC 2007, Kyoto, Japan, September 10-14, 2007, Proceedings*. IEEE Computer Society, 2007.

[10] D. Dhungana, R. Rabiser, P. Grünbacher, K. Lehner, and C. Federspiel. Dopler: An adaptable tool suite for product line engineering. In *Software Product Lines, 11th International Conference, SPLC 2007, Kyoto, Japan, September 10-14, 2007, Proceedings. Second Volume (Workshops)*. Kindai Kagaku Sha Co. Ltd., Tokyo, Japan, 2007.

[11] M. Fowler. Language workbenches: The killer-app for domain specific languages?, 2005.

[12] A. S. Gokhale, K. Balasubramanian, A. S. Krishna, J. Balasubramanian, G. Edwards, G. Deng, E. Turkay, J. Parsons, and D. C. Schmidt. Model driven middleware: A new paradigm for developing distributed real-time and embedded systems. *Science of Computer Programming*, 73(1), 2008.

[13] W. Jirapanthong and A. Zisman. Supporting product line development through traceability. In *12th Asia-Pacific Software Engineering Conference (APSEC 2005), 15-17 December 2005, Taipei, Taiwan*. IEEE Computer Society, 2005.

[14] L. C. L. Kats and E. Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In W. R. Cook, S. Clarke, and M. C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 444–463. ACM, 2010. (best student paper award).

[15] M. MendonÃ§a, A. Wasowski, and K. Czarnecki. Sat-based analysis of feature models is easy. In D. Muthig and J. D. McGregor, editors, *Software Product Lines, 13th International Conference, SPLC 2009, San Francisco, California, USA, August 24-28, 2009, Proceedings*, volume 446 of *ACM International Conference Proceeding Series*. ACM, 2009.

[16] J. Oldevik and Øystein Haugen. Higher-order transformations for product lines. In *Software Product Lines, 11th International Conference, SPLC 2007, Kyoto, Japan, September 10-14, 2007, Proceedings*. IEEE Computer Society, 2007.

[17] F. Roos-Frantz. A preliminary comparison of formal properties on orthogonal variability model and feature models. In D. Benavides, A. Metzger, and U. W. Eisenecker, editors, *Third International Workshop on Variability Modelling of Software-Intensive Systems, Seville, Spain, January 28-30, 2009. Proceedings*, volume 29 of *ICB Research Report*. Universitat Duisburg-Essen, 2009.

[18] M. Voelter. Best practices for dsls and model-driven development. *JOT*, 2009.

[19] M. Voelter and I. Groher. Handling variability in model transformations and generators. In *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM'07)*, Computer Science and Information System Reports, 2007.

[20] M. Voelter and K. Solomatov. Language modularization and composition with projectional language workbenches illustrated with mps. 2010.

[21] M. Völter. Architecture as language. *IEEE Software*, 27(2):56–64, 2010.

[22] M. Völter and I. Groher. Product line implementation using aspect-oriented and model-driven software development. In *Software Product Lines, 11th International Conference, SPLC 2007, Kyoto, Japan, September 10-14, 2007, Proceedings*. IEEE Computer Society, 2007.

[23] J. White, D. Benavides, D. C. Schmidt, P. Trinidad, B. Dougherty, and A. R. Cortés. Automated diagnosis of feature model configurations. *Journal of Systems and Software*, 83(7):1094–1107, 2010.