Generative Metaprogramming

Salvador Trujillo
IKERLAN Research Centre
Mondragon, Spain
strujillo@ikerlan.es

Maider Azanza
ONEKIN Research Group
The University of the Basque Country
San Sebastian, Spain
maider_azanza@ehu.es

Oscar Diaz
ONEKIN Research Group
The University of the Basque Country
San Sebastian, Spain
oscar.diaz@ehu.es

Abstract

Recent advances in Software Engineering have reduced the cost of coding programs at the expense of increasing the complexity of program synthesis, i.e. metaprograms, which when executed, will synthesize a target program. The traditional cycle of configuring-linking-compiling, now needs to be supplemented with additional transformation steps that refine and enhance an initial specification until the target program is obtained. So far, these synthesis processes are based on error-prone, hand-crafted scripting. To depart from this situation, this paper addresses generative metaprogramming, i.e. the generation of program-synthesis metaprograms from declarative specifications. To this end, we explore *(i)* the (meta) primitives for program synthesis, *(ii)* the architecture that dictates how these primitives can be intertwined, and *(iii)* the declarative specification of the metaprogram from which the code counterpart is generated.

Categories and Subject Descriptors D.2.11 [*Software Architectures*]: Domain-specific architectures; D.2.13 [*Reusable Software*]: Domain engineering

General Terms Design

Keywords Generative Programming, Metaprogramming, Generative Metaprogramming, Software Product Lines, Feature Oriented Model-Driven Development

1. Introduction

A main insight of this work is that recent advances in Software Engineering have reduced the cost of coding programs at the expense of increasing the complexity of program synthesis, i.e. the process of coming up with the final program. Model Driven Development (MDD) and Software Product Lines (SPL) are two cases in point. MDD conceives metaprograms as a pipeline of model transformations [15]. Most popular transformations are top-down where the program is modelled in a platform independent way. Distinct program viewpoints can coexist, while are gradually made more concrete, reaching eventually a platform specific model. Although MDD brings important benefits (e.g. reduced costs, improved quality, reduced development time), now the synthesis of the final program becomes more complicated. The traditional cycle of configuring-linking-compiling, now needs to be enriched with additional transformation steps that move the model down to the code. The synthesis metaprogram becomes more complex.

On the other hand, SPLs aim at building a set of related products out of a common set of core artifacts. Unlike MDD, now the stress is not so much on the abstraction level at which the software is specified, but on conceiving programs as pre-planned variations from common artifacts. The synthesis of a program starts with the SPL's common artifacts where variation points are gradually instantiated to establish the features that will be eventually exhibited by the end program. This can be conceived as a refinement process where the generality of the artifacts are eventually concretized into a particular program [8]. However, the description of how a program is synthesized out of a set of artifacts is far from trivial. Again, program synthesis becomes more elaborated.

MDD and SPL are now well-established approaches for reuse. Both together will result in even more important gains. However, the description of the process to synthesize final programs out of models and features ends up in complex scripts (e.g. derivations and refinements are intertwined). As a case in point, we reported a combined use of these two techniques [35]. For this case, a typical script to realize a synthesis metaprogram consists of around 500 LOC of batch processes that use 300 LOC of ANT makefiles and 2 KLOC of Java code, taking around 4 person/day to complete. So far, these synthesis paths were based on laborious, hand-crafted scripting. This paper is about facilitating the task of program synthesis.

Generative programming is about metaprograms that synthesize other programs. Metaprogramming is the concept that program synthesis is a computation. This work describes an approach to generate metaprograms, which when executed, will synthesize a target program of a product line. We name this process *generative metaprogramming*. Our intention is to accelerate the development of metaprograms by generating them from abstract specifications. Doing so, a metaprogram is declaratively *specified* rather than programmatically implemented.

Specifically, we focus on two types of (meta) primitives. First , the *derivation* operator which is used for moving models down to a specific platform but without adding additional functionality. This operator has been extensively used in MDD practises. Second, the *composition* operator which leverages a given model with a meaningful functionality. Here, “meaningful” should be understood as meaningful to the end-user, and in this way, it correlates with the notion of feature in SPL.

Our aim is a generative approach to metaprogramming. To attain this, we explore *(i)* the meta primitives for program synthesis, *(ii)* the architecture that dictates how these primitives can be intertwined, and *(iii)* the declarative specification of the metaprogram from which the code counterpart is generated. These are the ideas behind the *GeneRative metaprOgramming for Variable structurE*

GPCE'07, October 1–3, 2007, Salzburg, Austria.
Copyright © 2007 ACM 978-1-59593-855-8/07/0010. . . $5.00

(GROVE) approach. GROVE demands a new generation of models and tools. The aim of GROVE is to eventually facilitate synthesis metaprogramming. We begin with a review of background.

2. Background

Model Driven Development MDD is an emerging paradigm for software construction that uses models to specify systems, and model transformations to synthesize executables [15]. Program specification in MDD uses one or more models to define a target program. Ideally, these models are *platform independent (PIM)*. Model *derivations* convert platform independent models to *platform specific models (PSM)*, where assorted technology bindings are introduced. Possible results of transforming PIMs can be an executable or an input to an analysis tool, where both are themselves considered models.

Feature Oriented Programming FOP is a paradigm for creating software product lines where customized programs are synthesized by composing features [8]. Hence, features sketch not only increments in program functionality, but are really the building blocks of programs. An *FOP model* of a product line offers a set of operations, where each operation implements a feature. We write $M = \{f, h, i, j\}$ to mean model M has operations or features f, h, i and j. FOP distinguishes features as *constants* or *functions*. Constants represent base programs. Functions represent *program refinements* that extend a program that is received as input. The design of a program is a named expression, e.g.: $prog_2 = i \bullet j \bullet h$. Program $prog_2$ has features h, j, and i.

Metaprogramming is the concept that program development is a computation[1]. Batory asserts that MDD is a metaprogramming paradigm [4]. That is, models are program values and transformations are program functions that map these values. Scripts that transform models into executables are *metaprograms* (i.e., programs that manipulate values that themselves are programs). For example, *ant* makefiles are metaprograms; the values handled by a makefile metaprogram are program files and the execution of a makefile can produce an executable [14]. An MDD process can be written as a makefile (metaprogram) whose input values are models of target programs, and whose output values are synthesis targets (examples of such metaprograms in [29]). The connection of FOP to metaprogramming and MDD is simple: FOP treats programs as values, and features are functions that map values. Next, we show how FOP and MDD can be integrated.

Feature Oriented Model Driven Development FOMDD is a blend of FOP and MDD that shows how products in an SPL can be synthesized in an MDD way by composing features to create models, and then transforming these models into executables [35]. FOMDD supports synthesis metaprogramming using scripting, which when executed, synthesizes individual systems. The creation of this scripting involves repetitive, time-consuming and cumbersome tasks. However, such metaprograms are not monolithic blocks, but consist of distinct parts representing different operations to synthesize a program. Next, we elaborate on metaprograms for FOMDD, which defines the process to synthesize individual systems (i.e. how artifacts are assembled to yield an outcome product).

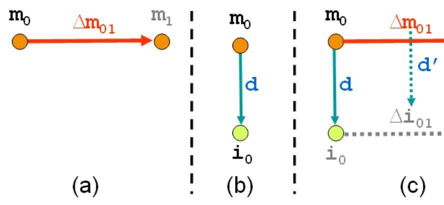


Figure 1. Synthesis Primitives

3. Architectural Metaprogramming

3.1 Synthesis Primitives

A synthesis metaprogram consists of basic primitives. This work does not attempt to provide an exhaustive list of those primitives but focuses on two of them, namely, model composition and model derivation.

Composition is used to synthesize programs in *Feature Oriented Programming (FOP)* [8]. This primitive composes a *base* model with model refinements to output a model with enhanced functionality (normally, exhibiting a new feature). Figure 1a sketches an example for composition where nodes represent models: m_0 (initial) and m_1 (extended). The edge stands for the **c***ompose* primitive parameterized with refinement $\triangle m_{01}$. Note that m_1 is synthesized as a result of composition. The expression for this primitive is: $m_1 = \triangle m_{01}.compose(m_0)$. (Although we write the composition of A and B as $A.compose(B)$, it can be represented alternatively as expression $A \bullet B$.)

Derivation is used for moving models down to specific platforms. Unlike composition, derivation adds details about the realization of the functionality but the functionality itself is preserved. Figure 1b shows an example of a derivation primitive where nodes represent models m_0 (source) and i_0 (target), and the edge stands for the **d***erive* primitive. Note that i_0 is synthesized as a result of the derivation. The expression for this primitive is: $i_0 = m_0.derive()$.

Composition and Derivation can be intermingled. Figure 1c shows an example of this combination. Nodes represent models where m_0 is an input and its remaining (m_1, i_0, i_1) are synthesized. Edges represent either composition primitives (e.g., horizontal edge with $\triangle m_{01}$ represents a composition with a *compose* refinement) or derivation primitives (e.g., vertical edge d stands for *derive*)[2].

The combined use of composition and derivation exposed a fundamental **commuting** relationship: *(1)* the composition of derived models equals *(2)* the derivation of a composed model. This implies that there are two alternative paths to synthesize i_1.

$$i_1 = (\triangle m_{01}.derive'()\,).compose\,(m_0.derive()\,) \quad (1)$$
$$i_1 = (\triangle m_{01}.compose(m_0)\,).derive() \quad (2)$$

When i_1 is the same through *(1)* and *(2)*, commuting holds. This means that composition and derivation are structure-preserving primitives.

3.2 Synthesis Architecture

Software architectures represent the design for describing the main parts of a software system. Traditionally, software architectures

[1] Metaprogramming is the writing of programs that write or manipulate other programs (or themselves). Excerpt from http://en.wikipedia.org/wiki/Metaprogramming.

[2] Note that a new edge d' appears. This edge represents a different *derive'* from edge $\triangle m_{01}$ to edge $\triangle i_{01}$. Remember d was from node to node previously. This d' is a model refinement-to-model refinement transformation.

have been considered as a set of interrelated components and connectors [2, 25].

The software architecture of a system is the *structure of the system*, which comprises software elements, the relationships between them, and the externally visible properties of those elements. Likewise, a **synthesis architecture** includes *(i)* software models, *(ii)* derivation and composition relationships between them and *(iii)*, structural properties such as the commuting one. The definition of this architecture is also referred to as *architectural metaprogramming* [6, 33].

3.3 Example

A synthesis architecture abstracts several primitives to yield an individual program out of a product line (e.g. a typical example consists of distinct derivations to synthesize executables from input models). A synthesis architecture aims to define the synthesis space for a given domain.

To illustrate these ideas, we introduced *PinkCreek*, which is a *product line of portlets* (building blocks of web portals), providing flight reservation capabilities to different portals. PinkCreek consists of up to 20 features, yielding hundreds of distinct individual Portlet products (refer to [13, 29] for details).

The PinkCreek product line was developed based on FOMDD, intermingling the notions of features from FOP and models from MDD. PMDD is a model-driven approach for Portlets, which specifies a portlet as a set of models from which its implementation can be derived [29]. We used PMDD to create PinkCreek.

According to the previous definition, the synthesis architecture for the PinkCreek case, includes:

- nodes (see Figure 2a): the upper-most node represents a statechart model m_{sc}. The nodes on the bottom are implementation artifacts: m_{jak} represents *Jak* classes and m_{jsp} JSP pages. The nodes in between are *Portlet Specific Language* models: m_{ctrl} defines the portlet controller, m_{act} contains the portlet actions, and m_{view} specifies the portlet views.
- edges: downward edges stand for derivation (e.g. $d_{sc2ctrl}$ is a derivation from m_{sc} to m_{ctrl}) whereas horizontal edges correspond to compositions (e.g., $\triangle m_{sc}$). Figure 2b extends Figure 2a with composition edges. The example focuses on how the incorporation of a single feature percolates along the distinct layers of the derivation structure. The support of additional features will rise homomorphic structures.
- structural properties: the commuting property should hold.

This exploration of the synthesis architecture paves the way to the generation of synthesis metaprograms.

4. Generative Metaprogramming

Generative Programming is the process to generate programs where automated source code creation is done through code generators to improve programmer productivity [12]. Our work goes one step beyond. Starting from a synthesis architecture specification, our aim is to generate a synthesis metaprogram, which is used to generate a program. We name this process *Generative **Meta**programming*.[3]

4.1 Overview

This work does not focus on the creation of the SPL capability. We consider that the SPL has been already created using existing approaches [8, 11]. SPL changes the individual product-centric development by introducing a general distinction between domain and application engineering. Domain engineering determines the commonality and the variability of the SPL, whereas application engineering synthesizes individual applications from the SPL. This SPL separation also holds for Generative Metaprogramming, but it additionally introduces an initial activity for MDD.

1. **Template Definition.** The architectural style is defined by specifying the metaprogram templates related to MDD development during domain engineering. Figure 2a shows the style for our PMDD domain. In general, this PMDD-template architecture is valid for any Portlet (i.e. each MDD domain has different templates). This style mainly sets the different abstraction layers through which a portlet specification goes until code artifacts are obtained.
2. **Generic Architecture.** The synthesis architecture specification is defined at the level of the product line. Features realizing an SPL are defined in terms of instances of MDD-templates created previously (i.e. each feature is defined as a template instance). This step belongs as well to domain engineering (though omitted in Figure 2).
3. **Specific Architecture.** A target program is typically expressed by its compound features in application engineering. From this feature choice, the program-specific synthesis architecture can be depicted connecting the (feature) template instances. Figure 2b shows such synthesis architecture for a PinkCreek program of two features. This program-specific synthesis architecture is traversed to generate a program-specific metaprogram, from which a program is synthesized. Figure 2c shows a path that defines a synthesis metaprogram, which synthesizes a PinkCreek program. The point is that from this path, the synthesis metaprogram code is generated.

4.2 Defining the Templates for Synthesis Architecture

Templates A synthesis architecture is basically formed by instances of two templates: constant and refinement feature templates.

Constant template contains the set of primitives from which the implementation of a constant or base program is derived. It represents derivation synthesis without composition (e.g. PMDD to develop a Portlet). Figure 3a shows this where nodes represent different models and edges represent derivations and local compositions.

Refinement template represents the refinement of a constant template in order to realize a feature (i.e., the synthesis a feature realizes). Figure 4a shows this situation where a node represents a model that can be composed with an edge representing a model refinement. The result of composition yields another node representing a model. Hence, the edges represent model (or code) refinements and *edge-to-edge* derivations represent (delta) model refinement transformations. To simplify this representation, edges are represented as nodes, and edge-to-edge derivations as edges. Figure 4b shows this simplification of Figure 4a.

A constant template corresponds to a constant feature, whereas a refinement template corresponds to a feature that extends a constant (i.e., refinement templates are repeatable). The point to highlight is that synthesis architecture of Figure 2b can be generated combining constant and refinement templates together. To specify the synthesis architecture, a graphical language is needed.

Specification There are currently several graphical specification languages (e.g., SVG[4]). *Graph eXchange Language* (GXL) has been selected to depict our synthesis architectures graphically because it provides a meta-model, language, and tool support [17, 28].

[3] More information on our approach to metaprogramming can be found at *http://www.onekin.org/grove/*

[4] SVG (Scalable Vector Graphics). http://www.w3.org/Graphics/SVG/

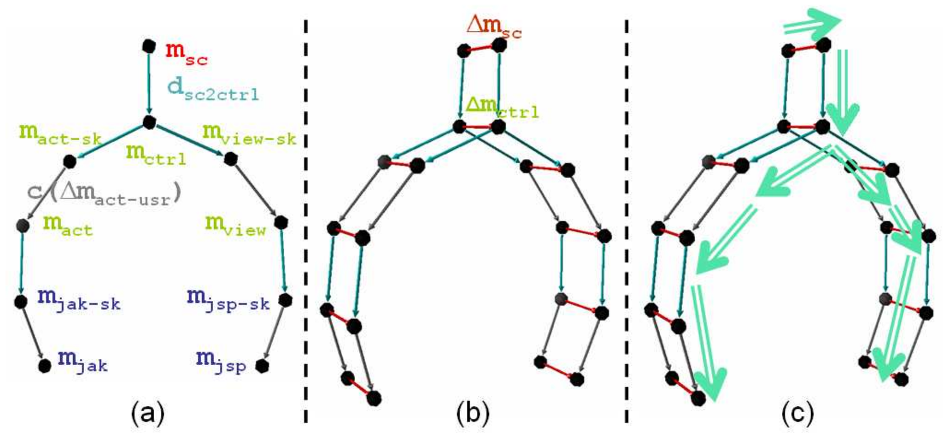
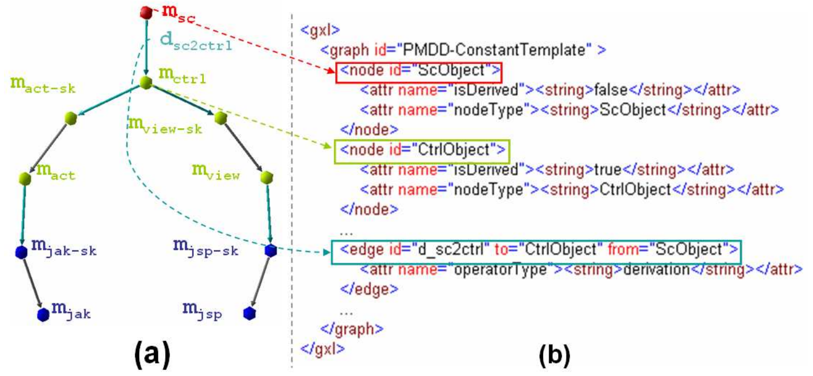
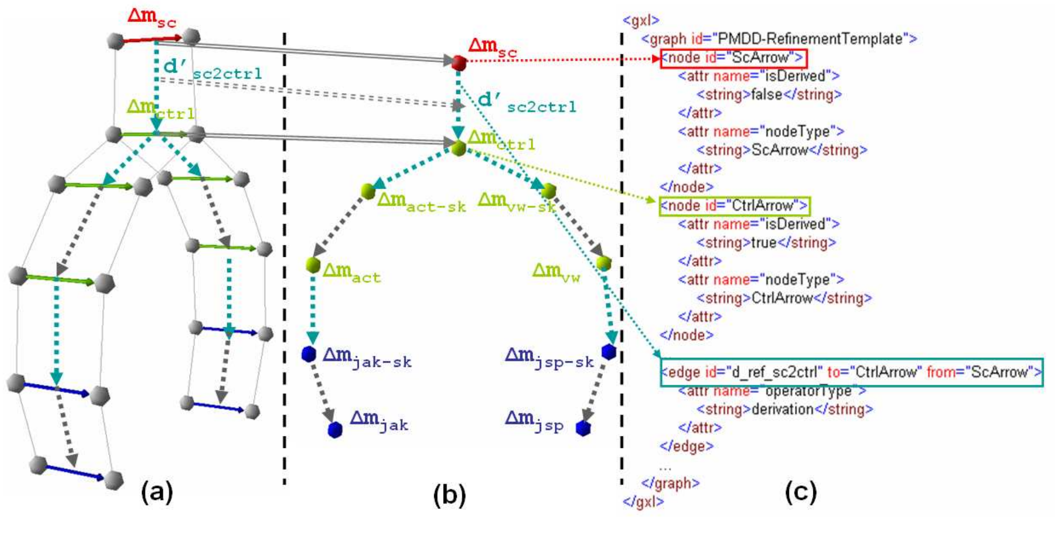


Figure 2. Synthesis Architecture for PinkCreek Example



```
<gxl>
  <graph id="PMDD-ConstantTemplate" >
    <node id="ScObject">
      <attr name="isDerived"><string>false</string></attr>
      <attr name="nodeType"><string>ScObject</string></attr>
    </node>
    <node id="CtrlObject">
      <attr name="isDerived"><string>true</string></attr>
      <attr name="nodeType"><string>CtrlObject</string></attr>
    </node>
    ...
    <edge id="d_sc2ctrl" to="CtrlObject" from="ScObject">
      <attr name="operatorType"><string>derivation</string></attr>
    </edge>
    ...
  </graph>
</gxl>
```

Figure 3. PMDD Constant Template Specification



```
<gxl>
  <graph id="PMDD-RefinementTemplate">
    <node id="ScArrow">
      <attr name="isDerived">
        <string>false</string>
      </attr>
      <attr name="nodeType">
        <string>ScArrow</string>
      </attr>
    </node>
    <node id="CtrlArrow">
      <attr name="isDerived">
        <string>true</string>
      </attr>
      <attr name="nodeType">
        <string>CtrlArrow</string>
      </attr>
    </node>
    <edge id="d_ref_sc2ctrl" to="CtrlArrow" from="ScArrow">
      <attr name="operatorType">
        <string>derivation</string>
      </attr>
    </edge>
    ...
  </graph>
</gxl>
```

Figure 4. PMDD Refinement Template Specification

GXL enables to specify both templates from which the synthesis architecture is formed.

GXL metamodel is designed to specify directed graphs [17]. Figure 3b shows a fragment of the GXL code that specifies the constant template of Figure 3a. This code is actually the representation of a model conforming the GXL metamodel. This figure was sketched using existing tool support [17, 28]. This tool produces the snippet of Figure 3b, which lists a fragment of code with two nodes and one edge connecting them. Nodes specify models m_{sc} and m_{ctrl}, and edge specifies derivation primitive $d_{sc2ctrl}$. In general, this code fragment represents the expression $m_{ctrl} = m_{sc}\,.d_{sc2ctrl}()$.

The elements of Figure 3b require additionally the specification of some attributes (e.g., name, type, primitive, whether is derived from other, etc). Nodes and edges have slightly different attributes. The attribute *nodeType* determines the type of a node. The attribute *isDerived* states whether a node is derived from other node. The attribute *operatorType* determines the primitive of an edge. This allows later to specify how to handle (edge) primitives over (node) elements. This extra information of the specification would be required later to generate template specific implementation.

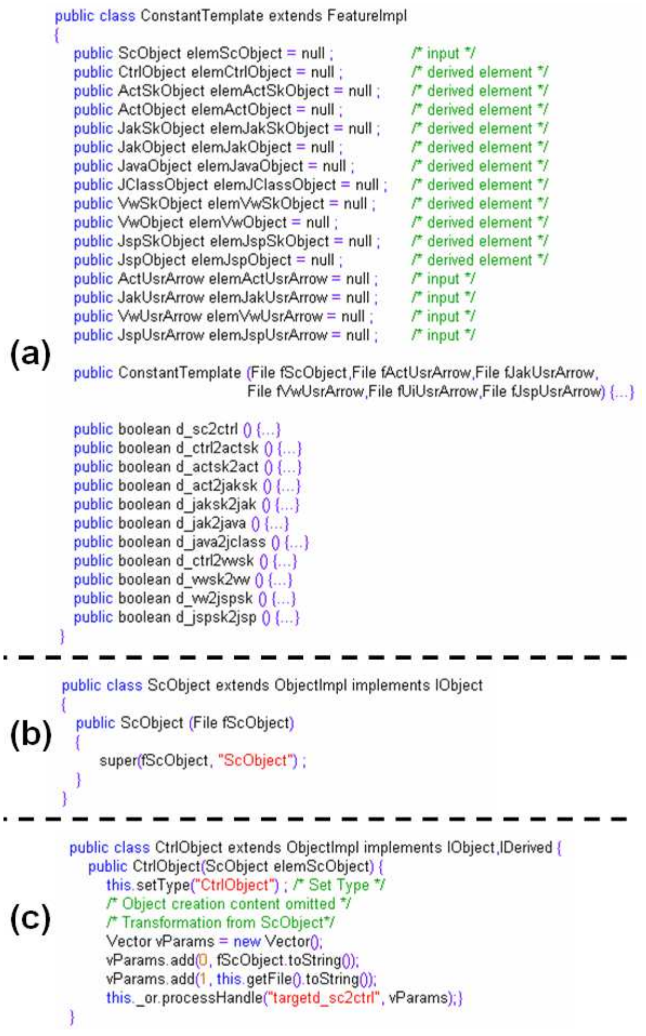
Likewise, Figure 4c shows partially the code snippet of a refinement template. This code fragment represents the expression $\triangle m_{ctrl} = \triangle m_{sc}\,.d'_{sc2ctrl}()$. The specification is similar to Figure 3b, but note that types of nodes are not model constants, but model refinements, and the edge is a transformation between such model refinements. This specification differs from the constant template because the actual way to operate it also differs.

Transforming from Specification to Implementation The next goal is to create an implementation to handle each element of the templates (i.e., node or edge). This implementation is the result of a model transformation from the specification of the two templates.

The result of the transformation is a set of *Java* classes that realize the synthesis architecture templates and give semantics to their primitives. The generated classes represent *(i)* the objects contained in the templates (for each node a class is generated), and *(ii)* the templates themselves (for each template a container-like class is generated).

First, a *Java* class is created for each node in a template. These classes represent nodes that are used later on by metaprograms. Some nodes are inputs and others are derived (those that depend on the input from another model). Figure 5b shows a snippet where an example for an input is shown. *ScObject* class encapsulates the synthesis functionality for models of type m_{sc}. As this is an input model, the functionality is reduced to create a handle for this file. Note that our framework provides a default implementation for the constructor (*ObjectImpl* in Figure 5b). The way to construct derived nodes is by creating constructors in their respective classes. Figure 5c shows an example for a derived class. *CtrlObject* class encapsulates the synthesis functionality for m_{ctrl}. Note that a constructor is used to derive m_{ctrl} from m_{sc}. Currently, this constructor invokes an *ant* target that performs the actual transformation and should be defined elsewhere. As composition is always similar, all node classes are composable (e.g., m_{ctrl} can be composed with $\triangle m_{ctrl}$). We provide a default implementation for artifact composition (*ObjectImpl* in Figure 5b). Similarly, each node element has a corresponding class.

Second, a *Java* class is created for each template that form the synthesis architecture (see Figure 5a). Actually, two classes are created: one for constant template and other for refinement template. These classes contain the objects that form each template. Figure 5a shows a snippet where an example for *PMDDConstantTemplate* is shown. This template contains references to its constituent parts (*ScObject*, *CtrlObject*, and so on). The template classes contain also one method for each edge of the template (*d_sc2ctrl*,


(a)
```
public class ConstantTemplate extends FeatureImpl
{
   public ScObject elemScObject = null ;               /* input */
   public CtrlObject elemCtrlObject = null ;           /* derived element */
   public ActSkObject elemActSkObject = null ;         /* derived element */
   public ActObject elemActObject = null ;             /* derived element */
   public JakSkObject elemJakSkObject = null ;         /* derived element */
   public JakObject elemJakObject = null ;             /* derived element */
   public JavaObject elemJavaObject = null ;           /* derived element */
   public JClassObject elemJClassObject = null ;       /* derived element */
   public VwSkObject elemVwSkObject = null ;           /* derived element */
   public VwObject elemVwObject = null ;               /* derived element */
   public JspSkObject elemJspSkObject = null ;         /* derived element */
   public JspObject elemJspObject = null ;             /* derived element */
   public ActUsrArrow elemActUsrArrow = null ;         /* input */
   public JakUsrArrow elemJakUsrArrow = null ;         /* input */
   public VwUsrArrow elemVwUsrArrow = null ;           /* input */
   public JspUsrArrow elemJspUsrArrow = null ;         /* input */

   public ConstantTemplate (File fScObject,File fActUsrArrow,File fJakUsrArrow,
                            File fVwUsrArrow,File fUiUsrArrow,File fJspUsrArrow) {...}

   public boolean d_sc2ctrl () {...}
   public boolean d_ctrl2actsk () {...}
   public boolean d_actsk2act () {...}
   public boolean d_act2jaksk () {...}
   public boolean d_jaksk2jak () {...}
   public boolean d_jak2java () {...}
   public boolean d_java2jclass () {...}
   public boolean d_ctrl2vwsk () {...}
   public boolean d_vwsk2vw () {...}
   public boolean d_vw2jspsk () {...}
   public boolean d_jspsk2jsp () {...}
}
```
(b)
```
public class ScObject extends ObjectImpl implements IObject
{
  public ScObject (File fScObject)
  {
      super(fScObject, "ScObject") ;
  }
}
```
(c)
```
public class CtrlObject extends ObjectImpl implements IObject,IDerived {
   public CtrlObject(ScObject elemScObject) {
      this.setType("CtrlObject") ; /* Set Type */
      /* Object creation content omitted */
      /* Transformation from ScObject*/
      Vector vParams = new Vector();
      vParams.add(0, fScObject.toString());
      vParams.add(1, this.getFile().toString());
      this._or.processHandle("targetd_sc2ctrl", vParams);}
 }
```
Figure 5. Generated Code Fragments

d_ctrl2actsk, and so on) that enables to construct derived nodes. Additionally, as the composition of templates is always similar, all template classes are composable to yield compound synthesis architectures. We provide a default implementation for this (*FeatureImpl* in Figure 5a). In general, a constant template contains model objects and model transformation methods, whereas a refinement template contains model refinement objects and their transformation methods.

As a result of these transformations, we obtain a class per node (with object functionality) where edge functionality is included and a class per each template (containing object references and edge methods). All these generated classes are later used by the actual synthesis metaprograms. This reuse is the reason behind the use of templates.

Tool Support is required for the domain engineer to specify the synthesis architecture templates. *GROVE Tool Suite* (GTS) is a set of tools we created to support GROVE. First, the templates are specified using *Graph eXchange Language* (GXL). Existing tools can be used by the domain engineer to draw such templates [28]. Second, such template specifications need to be transformed into the implementation code, which is used later on by the metaprogram. Thus, a code generator has been created (a.k.a., GTS code generator). Third, the semantics of some primitives needs to be specified completely (e.g., we use *ant* to specify how to handle them). This is introduced by GTS primitive handler.

4.3 Defining the Generic Synthesis Architecture

A program consists typically of a selected set of features realizing its increments in functionality. An SPL consists of the entire set of available features.

The templates introduced before enable to define constant and refinement features. The synthesis architecture can be specified by instantiating such templates for each feature. This involves defining only the set of input models that realize each feature. A generic synthesis architecture is then formed by the set of such feature templates.

Consider our example, PinkCreek product line consists of up to 20 features (e.g. *base*, *seat*, *checkin*, *assistance,* etc):

- As base is a constant feature, it is realized by instantiating a constant template. Figure 6 shows the instantiation of a constant template where the specific input models realizing such feature are specified.
- As *seat* is a refinement feature, it is realized by instantiating a refinement template. Figure 7 shows the instantiation of a refinement template where the specific input models realizing such feature are specified.

4.4 Defining the Program-Specific Synthesis Architecture

An individual program is a compound of features. Hence, the program-specific synthesis architecture is a compound of feature templates (i.e. template instances are the constituent parts of a program-specific synthesis architecture). We elaborate on how program-specific synthesis architectures are created and then on how metaprograms are synthesized. Figure 8a renders the synthesis architecture for a program of four features (e.g. *base*, *seat*, *checkin*, *assistance,* etc).

Specification A synthesis path represents how to synthesize a program traversing a synthesis architecture by means of composition and derivation. The fundamental idea is that the synthesis architecture should be specified before drawing the synthesis path (see Figure 8a).

Figure 8a shows the path for a PinkCreek program of four features. To specify this synthesis path, a specification is necessary to deal not only with feature composition, but as well with model derivation.

Existing specifications for program synthesis (e.g., AHEAD equation) do not consider other primitives but composition. An equation only specifies the set of features that distinguishes the program [8]. Figure 9a shows an example of this where features are listed in composition order. This would produce a synthesis geometry of one base substructure for *base* feature and one refinement substructure for *seat* feature. The expression is as follows $seat.compose\,(base)$. However, it lacks the derivation direction.

Hence, equation should be enhanced to introduce such extra information. Doing so, it introduces other primitives besides feature composition where composition and derivation can be used together. Figure 9b shows an example where *dsc2ctrl* derivation primitive is introduced after composition. Although this example is simple, it enables the equation to deal with derivation as well. The equation is as follows $(seat.compose\,(base)).derive_{sc2ctrl}()$.

Figure 9c is a similar example where the equation differs. The equation $(seat.derive'_{sc2ctrl}()).compose\,(base.derive_{sc2ctrl}()\,)$ represents the path in the figure. Figure 9c is simplified, it omits $d'_{sc2ctrl}$. The product synthesized via Figure 9b and 9c is the same (recall commuting). However, the synthesis time is different [35].

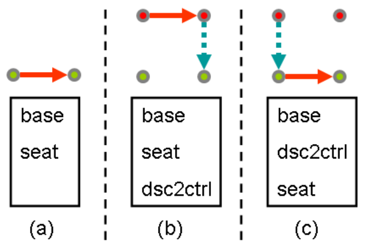


Figure 9. Path Specification Example

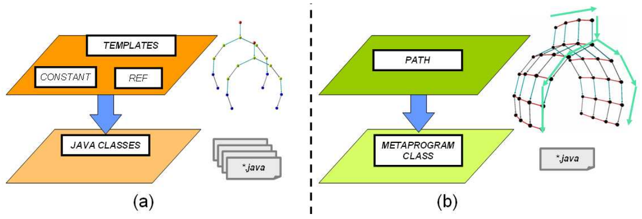


Figure 10. Big Picture

Transforming from Specification to Implementation The final goal is to synthesize a metaprogram (i.e., specifying how the program-specific synthesis architecture is traversed to get a metaprogram). This is obtained from the synthesis path specification as the result of applying a model transformation where the path specification is the input and the metaprogram implementation is the output. This transformation creates a specific metaprogram for each synthesis path in order to synthesize an end program.

As we define previously the generic objects to handle the synthesis architecture, the transformation is straightforward. For each feature template in the synthesis path, an object is created in the metaprogram to handle such template instance. Then, the synthesis path is traversed as follows. For each feature composition edge, a feature composition is called. For each derivation edge, a derivation is invoked.

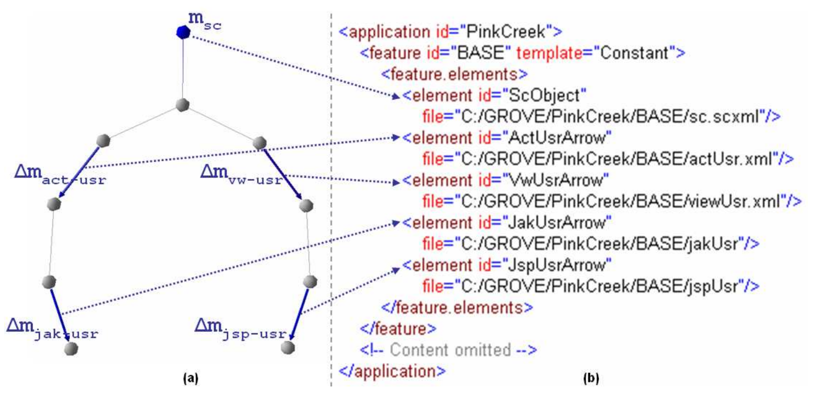
Consider our program of four features: *base*, *seat*, *checkin* and *assistance*. Figure 8a shows its program-specific synthesis architecture and one synthesis path that traverses it to synthesize a metaprogram. From such path specification, a Java class implementing the metaprogram is derived as follows. First, one template object per feature is created. Second, the horizontal path is traversed (i.e., feature templates are composed). Then, the vertical path is traversed (i.e., derivations are applied to the result to derive implementation artifacts). Figure 8b lists a fragment of this code. This resulting metaprogram is directly executable to synthesize an end-program.

Tool Support is required by the application engineer to specify program synthesis. We created specific tooling to support the generation of synthesis path metaprograms. First, the features of the program are selected using existing GUI tools to select features [29]. Second, from the feature selection, we can plot the whole program-specific synthesis architecture from which the user could select a synthesis path (see 8a). This path is then translated to a Java executable metaprogram (see 8b). Another tool is needed to generate such executable metaprogram. Our tooling also invokes the metaprogram, from which the program is synthesized ultimately.

4.5 Recap and Perspective

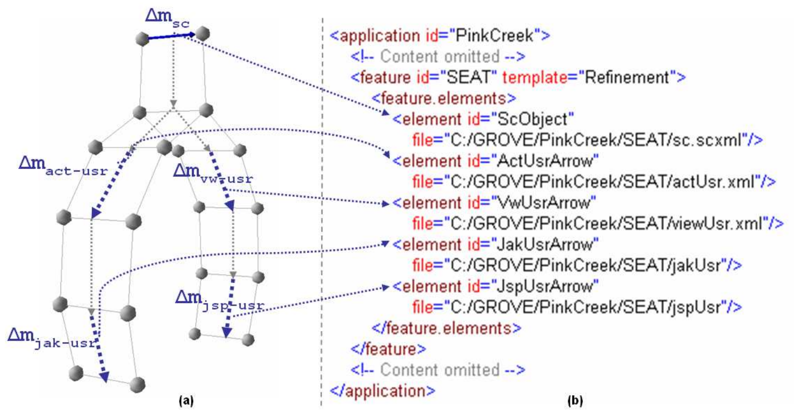
So far, we describe how to generate metaprogram code from a declarative specification be means of the PinkCreek example.

First, PinkCreek's synthesis architecture templates are specified (see Figure 10a). Actually, constant and refinement templates are



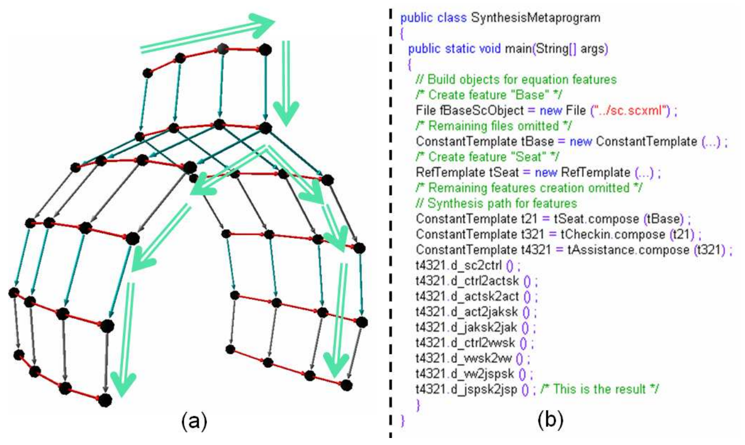
```
<application id="PinkCreek">
   <feature id="BASE" template="Constant">
      <feature.elements>
         <element id="ScObject"
            file="C:/GROVE/PinkCreek/BASE/sc.scxml"/>
         <element id="ActUsrArrow"
            file="C:/GROVE/PinkCreek/BASE/actUsr.xml"/>
         <element id="VwUsrArrow"
            file="C:/GROVE/PinkCreek/BASE/viewUsr.xml"/>
         <element id="JakUsrArrow"
            file="C:/GROVE/PinkCreek/BASE/jakUsr"/>
         <element id="JspUsrArrow"
            file="C:/GROVE/PinkCreek/BASE/jspUsr"/>
      </feature.elements>
   </feature>
   <!-- Content omitted -->
</application>
```

Figure 6. Template instance for *Base* Feature



```
<application id="PinkCreek">
   <!-- Content omitted -->
   <feature id="SEAT" template="Refinement">
      <feature.elements>
         <element id="ScObject"
            file="C:/GROVE/PinkCreek/SEAT/sc.scxml"/>
         <element id="ActUsrArrow"
            file="C:/GROVE/PinkCreek/SEAT/actUsr.xml"/>
         <element id="VwUsrArrow"
            file="C:/GROVE/PinkCreek/SEAT/viewUsr.xml"/>
         <element id="JakUsrArrow"
            file="C:/GROVE/PinkCreek/SEAT/jakUsr"/>
         <element id="JspUsrArrow"
            file="C:/GROVE/PinkCreek/SEAT/jspUsr"/>
      </feature.elements>
   </feature>
   <!-- Content omitted -->
</application>
```

Figure 7. Template instance for *Seat* Feature



```
public class SynthesisMetaprogram
{
  public static void main(String[] args)
  {
    // Build objects for equation features
    /* Create feature "Base" */
    File fBaseScObject = new File ("../sc.scxml") ;
    /* Remaining files omitted */
    ConstantTemplate tBase = new ConstantTemplate (...) ;
    /* Create feature "Seat" */
    RefTemplate tSeat = new RefTemplate (...) ;
    /* Remaining features creation omitted */
    // Synthesis path for features
    ConstantTemplate t21 = tSeat.compose (tBase) ;
    ConstantTemplate t321 = tCheckin.compose (t21) ;
    ConstantTemplate t4321 = tAssistance.compose (t321) ;
    t4321.d_sc2ctrl () ;
    t4321.d_ctrl2actsk () ;
    t4321.d_actsk2act () ;
    t4321.d_act2jaksk () ;
    t4321.d_jaksk2jak () ;
    t4321.d_ctrl2vwsk () ;
    t4321.d_vwsk2vw () ;
    t4321.d_vw2jspsk () ;
    t4321.d_jspsk2jsp () ; /* This is the result */
  }
}
```

Figure 8. Program-Specific Synthesis Architecture

specified (constant corresponds to Figure 3 and refinement corresponds to Figure 4). Model-to-code transformations are used to transform from these templates to the Java classes implementing them (classes are similar to those in Figure 5). Doing so, template classes are available to be used later on by each metaprogram class (e.g. while traversing a synthesis path).

Second, template instances are specified for the SPL features (i.e. for each feature, a template instance is defined with the models realizing it).

Third, the aim is to generate PinkCreek's programs (see Figure 10b). Specifically, we generate those synthesis metaprograms from which programs are synthesized. We start by selecting the program features (creating an equation). Doing so, the program-specific synthesis architecture of Figure 8a is rendered. A synthesis path

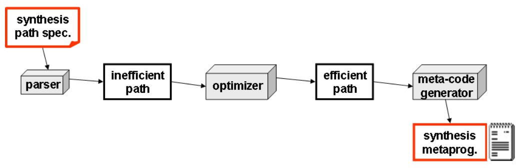


Figure 11. Generative Metaprogramming Process

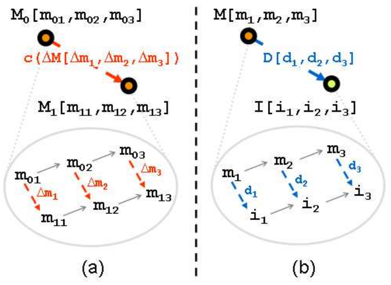


Figure 12. Feature Composition

specifies then how to traverse such architecture (e.g., Figure 8a shows an example of such traversal). From the synthesis path, a synthesis metaprogram code is derived using the model-to-code transformation we created (see Figure 8b). This metaprogram code uses the template classes and template instances created previously. Hence, the execution of this metaprogram ultimately leads to the synthesis of the program.

Process Figure 11 depicts the process to generate metaprograms. A number of challenges were addressed to turn it from an envision into a reality. This process is analogous to the SQL query evaluation process where *(i)* a synthesis path statement is specified, from this input *(ii)* the parser generates *(iii)* some inefficient path, then *(iv)* the efficiency of this path is evaluated by the optimizer, eventually *(v)* obtaining an efficient path, which is *(vi)* the input to the meta-code generator. The output of this process is *(vii)* the synthesis metaprogram code that is directly executable to synthesize a program.

Benefits GROVE automates significant and tedious tasks in generative metaprogramming. The benefit of this approach is that it is no longer necessary to create/modify the implementation of the synthesis metaprogram, but its specification (i.e., the way to specify synthesis is not coding, but drawing a graphical specification). Our approach reduces the development time of synthesis metaprograms and facilitates their evolution. For our PinkCreek case, 33 Java classes and 1.188 LOC are derived from a graphical input of two template files and 498 LOC. Likewise, synthesis path metaprograms are not coded, but derived from path specifications. For a typical metaprogram of 5 features, specified by an equation of 18 LOC, our tools produce an output metaprogram of 210 LOC. This metaprogram uses the 33 classes generated before. The number of LOC increases proportionally with the number of features: 380 LOC for 10 features (equation specification of 23 LOC); 550 LOC for 15; 720 LOC for 20; and 822 LOC for 23 (equation specification of 36 LOC). These figures pose a remarkable improvement from our previous scripting metaprograms.

5. Advanced Topics

5.1 Multiple Artifacts

The primitives presented before are applied to individual artifacts. However, primitives frequently involve more than one artifact. This

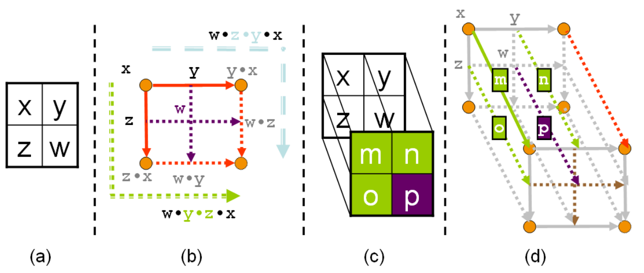


Figure 13. Origami Multi-dimensions

is the case of composition where multiple artifacts are simultaneously composed *á la* AHEAD [8]. A feature in AHEAD is implemented by multiple artifacts: introductions add new artifacts and refinements extend existing artifacts. Such artifacts can have multiple representations (e.g., code, makefiles, documentation, etc.).

Figure 12a shows an example of a simultaneous composition of multiple artifacts. Coarse-grained nodes M_0 and M_1 represent a set of artifacts (M_0 contains m_{01}, m_{02}, and m_{03}; M_1 contains m_{11}, m_{12}, and m_{13}). Likewise, edge $\triangle M$ consists of a set of artifacts ($\triangle m_1$,$\triangle m_2$,$\triangle m_3$) to refine the set of artifacts a node comprises. Note that M_1 is synthesized as a result of a composition: $M_1 = \triangle M.compose\,(M_0)$.

In general, coarse-grained nodes and edges contain the realization of a feature. Some features are base or constants (e.g., node M_0), some features are increments in functionality or functions (e.g., edge $\triangle M$), and a composition of features is a program (e.g., M_1). Similarly to composition, derivation comprises frequently multiple artifacts. Figure 12b shows an example of a derivation of multiple artifacts.

5.2 Multiple Product Lines

Synthesis is typically scoped to an individual product line. However, in certain contexts synthesis of multiple product lines may be desirable. Consider an example where different Portlets are combined together to synthesize a Portal. Although this setting is unlikely nowadays in SPL, it is common in service-oriented architectures. Future work should address how to automate synthesis involving external product lines. More to the point, this would impact on our synthesis architectures. Consequently, the architecture metaprogramming should also consider service-oriented synthesis architectures [30].

5.3 Multiple Paradigms

Our generative metaprogramming approach is restricted so far to FOMDD (i.e. it is a combination of software engineering productivity paradigms restricted to the blend of FOP and MDD). However, this combination is clearly open to other software engineering paradigms such as weaving of aspects in Aspect Oriented Programming (AOP) [1], refactoring of features [19, 34], interactions of features [20], harvesting of models [24], and others. We are currently facing the combination of more than two paradigms (e.g. FOP, AOP and MDD [21]).

5.4 Multiple Dimensions

The set of primitives presented so far configure a space of synthesis design (similar to composition design [5]). In general, this synthesis design consists of multiple dimensions, thus becoming more complex. So far, we come up with two types of dimensions: space and time.

Space and Origami: Multidimensions in space arise when there are orthogonal feature models. Origami deals with these complex

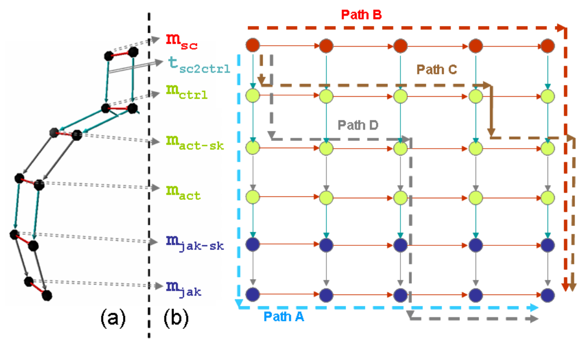


Figure 14. PinkCreek Optimization

relationships among orthogonal features by using matrices [7]. An Origami matrix is a n-dimensional matrix where each dimension is formed by a set of features and its cells are the feature modules that implement the functionality at the intersection of their coordinates in the n-dimensions. The benefit of Origami is that represents largely linear specifications of systems that have potentially exponential complexity. For the purpose of this work, note that Origami poses multiple *space* dimensions to be considered.

Figure 13a shows an example with a simple *2x2* matrix. Figure 13b sketches the synthesis architecture for this matrix, translating Origami matrix into such representation. The synthesis architecture becomes more complicated as the number of dimensions increase. Figure 13c shows a *2x2x2* matrix (i.e., a cube) and Figure 13d sketches its representation. Note that these examples only use composition, but not derivation. These figures illustrate how the synthesis architecture explodes with the number of dimensions.

The representation of multiple dimensions exposes some limitations of our work. It is not evident how to draw these synthesis architectures even for small dimensions, and it is almost impossible to depict out the commuting diagrams for n-matrices. Future work might address a generalization of an Origami matrix, that allows complex, multi-dimensional commuting diagrams to be expressed. In this future scenario, derivation should be represented using Origami as well.

Time and Evolution: The design spaces presented so far represent a static view of the synthesis architecture. However, as any software, a synthesis metaprogram also evolves over time, making the synthesis variable [33]. Remarkably, this evolution forces to consider also multiple *time dimensions* [31].

5.5 Multiple Paths

There are different possible paths while traversing a synthesis architecture design space to generate a synthesis metaprogram. Commuting only refers to two possible paths, but there exist additional paths. Figure 14 illustrates this situation where further paths are shown for PinkCreek synthesis. Figure 14a shows partially the synthesis. This program-specific synthesis architecture can be traversed by multiple synthesis paths to yield the same product (see Figure 14b).

Each synthesis path implies a different cost. A number of rules are required to optimize cost. These rules would be based on different optimization criteria (e.g., build-time, program size or quality). Under each criteria, one or more optimized path appears. This shortest path between two points is known as *geodesic*[5].

[5] A geodesic can be regarded also as the shortest spanning graph between an input set of points and an output set of points, being its resolution not straightforward. According to Batory, it involves solving the Directed Steiner-Tree problem, which is NP-hard [5, 9].

In our process of Figure 11, the optimizer would take the synthesis path selected by the user as input and would return a geodesic. To turn this vision into a reality we created tools to measure the cost of traversing each edge of the geometry. We have measured the cost of operation time and the program size. However, we do not yet solve the optimization problem. An alternative is the use of rules to find the geodesic. Synthesis optimization is a matter for future work.

6. Related Work

Feature Oriented Model Driven Development (FOMDD) is a blend of FOP and MDD where the feature-structure imposed by AHEAD is translated to MDD. GROVE is a model to support the generation of synthesis in FOMDD [35].

FOMDA stands for Feature Oriented Model Driven Architecture. Although the name is close to FOMDD, the issue addressed by FOMDA is different. FOMDA aims to specify the flow of transformations in MDA using a feature model [3]. Doing so, features are not used to denote increments in program functionality, but also model transformations.

AHEAD is an algebraic model for feature composition that structures SPL artifacts for composition [8]. Inspired by AHEAD, GROVE is focused on the architecture of synthesis metaprograms where the aim is the generative metaprogramming using abstract specifications. GROVE aims to introduce an architectural model for SPL synthesis. Actually, GROVE describes an initial algebraic representation, even though further work is needed to assess the mathematical implications of this representation [32]. The basic ideas behind model driven development pushed us to abstract the synthesis process. Actually, we first implemented those classes that later we attempted to model in order to generate them. Essentially, GROVE applies model-driven ideas to the generation of synthesis metaprograms.

A software factory is a particular approach to product-line development where a software factory schema specifies the artifacts involved in a factory, but not how such artifacts are synthesized [16]. GROVE additionally introduces the metaprogram generation for manipulating product-line artifacts.

Intentional programming is a collection of concepts which enable software source code to reflect the precise information, called an intention, which programmers have in mind when conceiving their work. Those intentions are actually abstractions [26, 27]. Similarly, GROVE introduces abstraction that enables the generation of synthesis code.

Increasingly complex systems require novel approaches where analogies with biological phenomena could be useful to inspire them (e.g., the study of structural biology to inspire GROVE templates) [22].

The architectures presented in this work are far from the *"perception-representation-action loops"* (a.k.a., *Design Animism*) presented by Laurel [18]. However, we believe the exploration of this aesthetic conception is worth with regard to our architectural styles.

Synthesis is also known as product production [23]. Production planning defines how programs are built. It provides the strategical vision of the production process [10]. GROVE specifically concentrates on the synthesis process of this managerial plan.

7. Conclusions

This work described ideas to synthesize metaprograms, which when executed, will synthesize a target program of a product line. Specifically, we elaborated on the generation of metaprograms from abstract specifications. To attain this, we presented the *GenеRative metaprOgramming for Variable structurE* (GROVE) as an approach to specify a synthesis architecture, from which a synthesis path can be specified in order to generate the code of a synthesis metaprogram. The execution of such metaprogram code synthesizes a target program of a product line. So far, a limited number of primitives is available mainly for composition and derivation.

GROVE called for a companion set of tools. *GROVE Tool Suite* (GTS) supports our generative approach to metaprogramming. GTS is realized by 26 classes and 4 KLOC Java. We implemented also three model transformations (two for synthesis architecture templates and one for synthesis path) realized by 600 LOC of XSL code. GTS encapsulates and reuses logic and infrastructure common to generative metaprogramming and provides the base functionality on top of which metaprogram-specific functionality is built.

Acknowledgments. We appreciate the always helpful insights and comments from Don Batory on earlier drafts of this paper. We thank him and the staff of *Shady Grove* restaurant for a pleasant and inspiring dinner. Felipe I. Anfurrutia also provides insightful comments to an early version of this work. This work was co-supported by the Spanish Ministry of Science & Education, and the European Social Fund under contract TIC2005-05610. Azanza enjoys a doctoral grant from the Basque Government.

References

[1] S. Apel, T. Leich, and G. Saake. Aspectual Mixin Layers: Aspects and Features in Concert. In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28*, 2006.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 2003.

[3] F.P. Basso, T. Cavalcante de Oliveira, and L. B. Becker. Using the FOMDA Approach to Support Object-Oriented Real-Time Systems Development. In *ISORC, Gyeongju, Korea*, pages 374–381, April 2006.

[4] D. Batory. Multi-Level Models in Model Driven Development, Product-Lines, and Metaprogramming. *IBM Systems Journal*, 45(3), 2006.

[5] D. Batory. From Implementation to Theory in Program Synthesis. In *Keynote at Principles of Programming Languages (POPL), Nice, France, Jan 17-19*, 2007.

[6] D. Batory. Program Refactoring, Program Synthesis, and Model Driven Development. In *Keynote at European Joint Conferences on Theory and Practice of Software (ETAPS) Compiler Construction Conference, Braga, Portugal, Mar 24 - Apr 1*, 2007.

[7] D. Batory, J. Liu, and J.N. Sarvela. Refinements and Multi-Dimensional Separation of Concerns. In *11th ACM Symposium on Foundations of Software Engineering (SIGSOFT), Helsinki, Finland, Sep 1-5*, 2003.

[8] D. Batory, J.Neal Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering (TSE)*, 30(6):355–371, June 2004.

[9] M. Charikar, C. Chekuri, T.Y. Cheung, Z. Dai, A. Goel, S. Guha, and M. Li. Approximation Algorithms for Directed Steiner Problems. *J. Algorithms*, 33(1):73–91, 1999.

[10] G. Chastek and J.D. McGregor. Guidelines for Developing a Product Line Production Plan. Technical report, CMU/SEI, June 2002. CMU/SEI-2002-TR-06.

[11] P. Clements and L.M. Northrop. *Software Product Lines - Practices and Patterns*. Addison-Wesley, 2001.

[12] K. Czarnecki and U. Eisenecker. *Generative Programming*. Addison-Wesley, 2000.

[13] O. Díaz, S. Trujillo, and S. Perez. Turning Portlets into Services: the Consumer Profile. In *16th International World Wide Web Conference (WWW 2007), Banff, Canada, May 8-12*, 2007.

[14] Apache Software Foundation. Apache Ant. http://www.ant.apache.org/.

[15] D.S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. Wiley, 2003.

[16] J. Greenfield, K. Short, S. Cook, and S. Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2004.

[17] R. C. Holt, A. Winter, and A. Schürr. GXL: Toward a Standard Exchange Format. In *7th Working Conference on Reverse Engineering (WCRE 2000), Brisbane, Australia*, November 2000.

[18] B. Laurel. Designed Animism: Poetics of a New World. In *Keynote at 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2006), Portland, Oregon, USA, October 22-26*, 2006.

[19] J. Liu, D. Batory, and C. Lengauer. Feature Oriented Refactoring of Legacy Applications. In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28*, 2006.

[20] J. Liu, D. Batory, and S. Nedunuri. Modeling Interactions in Feature Oriented Designs. In *International Conference on Feature Interactions (ICFI 2005), Leicester, United Kingdom, June 28-30*, June 2005.

[21] R.E. Lopez-Herrejon, D. Batory, S. Trujillo, and A. Cavarra. Towards a Unified Approach for Software Product Line Modelling with UML. In *Draft*, 2007.

[22] D.C. Marinescu and L. Bölöni. Biological Metaphors in the Design of Complex Software Systems. *Future Generation Computer Sciences*, 17(4):345–360, January 2001.

[23] J.D. McGregor. Product Production. *Journal Object Technology*, 3(10):89–98, November/December 2004.

[24] T. Reus, H. Geers, and A. van Deursen. Harvesting Software Systems for MDA-Based Reengineering. In *ECMDA-FA*, pages 213–225, 2006.

[25] M. Shaw and D. Garlan. *Software Architecture*. Prentice Hall, 1996.

[26] C. Simonyi. Intentional Programming: Innovation in the Legacy Age, 1996. Presented at IFIP Working group 2.1. Available from URL http://www.research.microsoft.com/research/ip/.

[27] C. Simonyi, M. Christerson, and S. Clifford. Intentional Software. *SIGPLAN Not.*, 41(10):451–464, 2006.

[28] Sourceforge.net. GXL (Graph eXchange Language). http://gxl.sourceforge.net/.

[29] S. Trujillo. *Feature Oriented Model Driven Product Lines*. PhD thesis, School of Computer Sciences, University of the Basque Country, March 2007. http://www.struji.com.

[30] S. Trujillo. Service Oriented Product Lines. In *Draft*, 2007.

[31] S. Trujillo, G. Aldekoa, and G. Sagardui. Tracking the Evolution of Feature Oriented Product Lines. In *XII Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2007), September 11-14*, 2007.

[32] S. Trujillo, M. Azanza, and O. Díaz. Endogenous and Exogenous Refinements in Concert. In *Draft*, 2007.

[33] S. Trujillo, M. Azanza, O. Díaz, and R. Capilla. Exploring Extensibility of Architectural Design Decisions. In *Second Workshop on SHAring and Reusing architectural Knowledge, Architecture, rationale and Design Intent (SHARK/ADI 2007) at 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, Minnesota (USA), May 19-20, 2007.*, 2007.

[34] S. Trujillo, D. Batory, and O. Díaz. Feature Refactoring a Multi-Representation Program into a Product Line. In *5th International Conference on Generative Programming and Component Engineering (GPCE 2006), Portland, Oregon, USA, Oct 24-27*, 2006.

[35] S. Trujillo, D. Batory, and O. Díaz. Feature Oriented Model Driven Development: A Case Study for Portlets. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, Minnesota, USA, May 20-26*, 2007.