Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

A Lightweight Modeling Approach Based on Functional Decomposition

Pierre Kelsen[a] Qin Ma[a] Christian Glodt[a]

a. Dept. of Computer Science, University of Luxembourg, Luxembourg

Abstract Creating models and transforming them using current MDE techniques is not easy: it generally requires mastering several non-trivial languages such as a metamodeling languages and a model transformation language. We propose a two-pronged approach for tackling language complexity for the case of model-to-text transformations. We first allow the user to define the metamodel in an example-driven fashion in which (s)he incrementally builds a set of examples and automatically infers the metamodel from them. The example-driven approach is based on a new object-modelling notation named OYAML that is both human- and machine- readable. Second we break down the complexity of writing the transformation itself by separately defining the functional decomposition of the transformation function using a new modelling language named FUDOMO. This will then allow the user to describe the precise behaviour in a general purpose programming language that (s)he is familiar with. Because they do not need to be very expressive, OYAML and FUDOMO are small languages when compared to commonly used metamodeling and model-to-text transformation languages. We provide a web-based tool, also named FUDOMO, that assists the user in this example-driven approach to model-to-text transformations and currently supports the use of Javascript and Python for defining the precise behaviour of model transformations.

Keywords functional decomposition, example-driven modeling, object models, model-to-text transformations, model transformation, code generation

1 Introduction

Model-driven engineering techniques can be challenging to use: they usually require the mastery of several non-trivial languages, e.g., a metamodeling language and a model transformation language. In the present paper we tackle this problem of language complexities in the context of the definition of model-to-text transformations.

In current practice the following steps are required for defining a model-to-text transformation: first a metamodel for the domain of interest has to be defined. Second

Pierre Kelsen, Qin Ma, Christian Glodt. *A Lightweight Modeling Approach Based on Functional Decomposition*. In Journal of Object Technology, vol. 19, no. 2, 2020, pages 15:1–22. doi:10.5381/jot.2020.19.2.a15

the transformation itself has to be written. There are currently two main approaches for the latter task:

1. Use a general purpose programming language like Java to query the model (using a suitable API) and produce the required output directly, or

2. Use a template-based approach for defining the structure of the output, and replace the dynamic parts (which are dependent on the model, as opposed to static parts, which do not depend on the model) by code snippets that produce the desired result

Approach 2 is generally preferred to approach 1 because it provides a cleaner separation of the static parts and the dynamic parts, resulting in a more understandable transformation definition. If we now want to apply a model-to-text transformation in a particular domain we first need to become familiar with a metamodeling framework, and secondly we need to learn a new transformation language (typically based on templates). Because both the metamodeling language and transformation language are quite expressive, they are generally not easy to master.

In this paper we strive to reduce the complexity of involved languages by combining two approaches: firstly the user directly specifies the examples using a simple object-modelling notation named OYAML that is based on the YAML language. From the set of examples, the user can infer, in an incremental fashion, the underlying metamodel. Secondly the definition of the transformation itself is broken down into two parts:

- we define the functional decomposition of the transformation function using a new language called FUDOMO, which allows us to specify the decomposition of the transformation function into progressively simpler functions until the point where the resulting functions can be trivially computed from the example models. This is done by formulating a set of decompositions, with each decomposition having a function on the left side and the functions it is decomposed into on the right side.

- Once these decompositions have been defined, the exact way in which a function is computed from the functions that it has been decomposed into is defined using a general purpose programming language that the user is familiar with.

We also provide a tool [10] that supports this example-driven approach to model-to-text transformations. The tool currently supports both Javascript and Python, two widely used programming languages, for specifying the computation of left-hand side functions in terms of right-hand side functions.

Because both OYAML and FUDOMO are not very expressive, they are rather small languages compared to their counterparts, the metamodeling language and the transformation language, in the traditional approach. By reusing a programming language that (s)he is already familiar with the total effort in learning new languages is reduced and thus the overall modelling effort should be more manageable.

This paper is structured as follows: in the next section we provide a guided tour of our approach with the help of a concrete example. In section 3 we present the OYAML language for writing object models. Functional decomposition and associated decomposition graphs are introduced in section 4. In section 5 we add polymorphism to our functional decomposition framework. The FUDOMO language for describing functional decomposition is given in section 6. We present a more complex example as a case study in section 7. In section 8 we present the overall process for designing

model-to-text transformations using our example-driven method. Our contributions are discussed and put into the context of related work in section 9. We offer concluding remarks in the final section.

2 Guided Tour

In this section we illustrate the main features of our approach with the help of an example. A more substantial example will be discussed in section 7. We shall use the term *object model* when talking about a concrete example model. This term will be more formally defined in section 4.

2.1 Writing Down the Object Model

To write down object models, we use a textual notation. We introduce a new modeling notation, called OYAML, that is based on YAML [9].

We use finite-state machines for our first example. The machines we use here have a finite set of states, as well as labeled transitions between states. Each finite-state machine has an initial state and a final state. Here is a simple example of a finite state machine written in OYAML:

```
- FSM:
    - name: EvenZeros
    - init >: s0
    - end >: s0
    - State s0:
        - name: s0
    - State  s1:
        - name: s1
    - Transition:
        - label: 0
        - source >: s0
        - target >: s1
    - Transition:
        - label: 0
        - source >: s1
        - target >: s0
    - Transition:
        - label: 1
        - source >: s0
        - target >: s0
    - Transition:
        - label: 1
        - source >: s1
        - target >: s1
```

In this example there is a single object of type FSM at the top level that has an attribute `name`. This attribute is followed by two references (indicated by the symbol "`>:`"): reference `init` refers to the initial state with id `s0`, and reference `end` refers to the final state also with id `s0`. The `FSM` object contains two objects of type `State`, each with an id (following the type name), and four objects of type `Transition` without ids. Containment is indicated via indentation.

```
Root:
  cont:
    - FSM
FSM:
  name:
    - String
  cont:
    - State
    - Transition
  end:
    - State
  init:
    - State
Transition:
  label:
    - String
  source:
    - State
  target:
    - State
State:
  name:
    - String
```

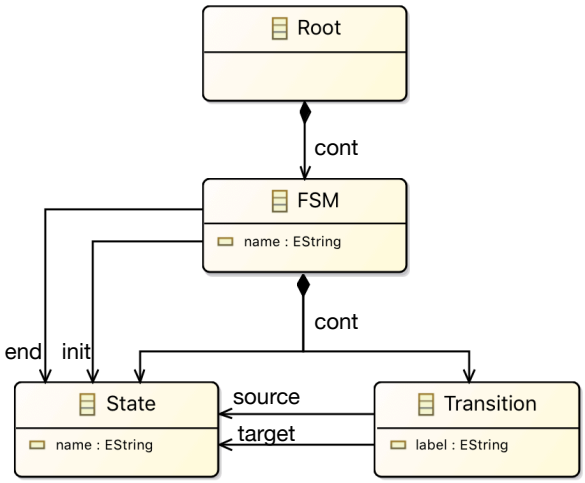

Figure 1 – Metamodel Inferred from the Example FSM

2.2 Inferring the Metamodel

The metamodel that can be inferred from the object model is presented in the Figure 1, with the textual representation (inferred by the FUDOMO tool) given on the left and the usual graphical representation in terms of a class diagram on the right. Two remarks need to be made: (1) There is an implicit `Root` object containing all top-level objects in each object model (i.e., those with lowest indentation level). (2) There is an implicit containment reference `cont`: the `Root` object contains all top-level objects (of type `FSM`) and an `FSM` object contains `State` and `Transition` objects.

2.3 Transforming the Example

From the above example we would like to generate a representation in the DOT language [3] so it can be easily visualized. Figure 2 shows on the left-hand side the textual output we want and on the right-hand side the visualization of it. We can view the transformation as a function that takes an `FSM` object model as input and returns its textual DOT-representation. We express this transformation function as a *typed function*, that is, a function that is defined in the context of a type. In this case the transformation will be defined in the context of the `FSM` type. Let us denote the transformation function by `FSM.fsm2dot`.

At the heart of our transformation approach is the notion of *functional decomposition*: we try to decompose typed functions into simpler constituent functions until these constituent functions can be trivially computed from the object model.

We can logically divide the output into three sections: a header section, a content section, and a footer section. For each section we define a related typed function. We

```
digraph {
  sinit [shape=point]
  s0 [peripheries = 2]
  sinit -> s0
  s1
    s0 -> s1 [label="0"]
    s1 -> s0 [label="0"]
    s0 -> s0 [label="1"]
    s1 -> s1 [label="1"]
}
```

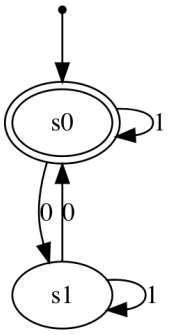

Figure 2 – Visual Representation of the `EvenZeros` Finite State Machine in DOT

obtain the decomposition expressed in the FUDOMO language as follows:

```
FSM.fsm2dot: header, content, footer
```

The right-hand side consists of three functions defined for the `FSM` type; the `FSM` type is omitted in the names of those functions since they share the same context type as the left-hand side.

Both the `header` and `footer` functions are constant functions, i.e., their value does not depend on the object model. For that reason they have an empty decomposition, represented in FUDOMO as:

```
FSM.header:
FSM.footer:
```

Next we note that `FSM.content` depends on the `State` and `Transition` objects it contains. We define functions `State.rep` and `Transition.rep` that produce the representation of the `State` and `Transition`, respectively, in DOT syntax. Thus we get the next decomposition:

```
FSM.content: cont -> State.rep, cont -> Transition.rep
```

Note the use of forward arrows which express navigation via containment reference. The value for State.rep depends on the name of the State, the init and final states of the containing FSM, and the identity of the State object expressed by the `center` keyword.[1] We express this by the following decomposition:

```
State.rep: name, cont <- FSM.init, cont <- FSM.end, center
```

Note the backward arrows from `cont` which express that this `State` has to follow the `cont` reference backwards to navigate to the containing `FSM`. Such backward arrows (later called *reverse edges*) are generally needed when relevant data is not accessible via forward references.

Finally the `Transition` representation, given by typed function `Transition.rep`, depends on the names of the `source` and `target` States as well as the `label` of the `Transition`. In FUDOMO we write this as:

```
Transition.rep: source -> State.name, target -> State.name, label
```

The full FUDOMO model thus contains six decompositions:

```
FSM.fsm2dot: header, content, footer
FSM.header:
FSM.content: cont -> State.rep, cont -> Transition.rep
```

[1] The corresponding function will be formally defined in section 4 (definition 8)

```
State.rep: name, cont <- FSM.init, cont <- FSM.end, center
Transition.rep: source -> State.name, target -> State.name, label
FSM.footer:
```

We can stop here since the decompositions have broken the original typed function `FSM.fsm2dot` down to the level of attributes and references, which can be directly obtained from the object model (without further calculations).

2.4 Implementing the Transformation Function

After having decomposed the `FSM.fsm2dot` function we still need to specify how to exactly compute the function from its constituent functions. The transformation approach we present in this paper is hybrid in the sense that it combines model and code: the model describes the decomposition of the transformation function and the code describes the exact computation. The model is expressed in the FUDOMO language. The code can we expressed in a number of common GPLs: currently Javascript and Python are supported by our tool [10].

We choose to implement the transformation function in JavaScript. For this the FUDOMO tool allows us to generate function headers, to be filled in with the missing code. Here is the resulting code with the manually added code highlighted:

```
module.exports = {
 /**
  * FSM.fsm2dot:
  * @param header The "header" of this FSM
  * @param content The "content" of this FSM
  * @param footer The "footer" of this FSM
  */
 FSM_fsm2dot: function(header, content, footer) {
   return header + content + footer;
 },
 /**
 * FSM.content
 * @param cont_State_rep {Array} The sequence of "rep" values of State
 * objects contained in this FSM
 * @param cont_Transition_rep {Array} The sequence of "rep" values of
 * Transition objects contained in this
 * FSM
 */
FSM_content: function(cont_State_rep, cont_Transition_rep) {
   return ' ' + cont_State_rep.join('\n ') + cont_Transition_rep.join('\n ')
        + '\n';}}
 },
 /**
  * State.rep:
  * @param name The "name" of this State
  * @param _cont_FSM_init {Set} The set of "init" values of FSM objects that
  * contain this State
  * @param _cont_FSM_end {Set} The set of "end" values of FSM objects that
  * contain this State
  * @param center This State
  */
 State_rep: function(name, _cont_FSM_init, _cont_FSM_end, center) {
   let s_rep = '';
   if (_cont_FSM_end.has(center)){
```

```
      \textbf{s_rep = name + ' [peripheries = 2]';}
  \textbf{} else {}
      \textbf{s_rep = name;}
  \textbf{}}}
  let re = '';
  if (_cont_FSM_init.has(center)){
   re = ' sinit [shape=point]\n'
    + ' ' + s_rep + '\n'
    + ' sinit -> ' + name;
  } else {
   re = s_rep;
  }
  return re;
 },
 /**
  * Transition.rep:
  * @param source_State_name {Array} The sequence of "name" values of State
  * objects referred to by attribute "source" in this Transition
  * @param target_State_name {Array} The sequence of "name" values of State
  * objects referred to by attribute "target" in this Transition
  * @param label The "label" of this Transition
  */
 Transition_rep: function(source_State_name, target_State_name, label) {
  return ' ' + source_State_name + ' -> ' + target_State_name + ' [label=\"'
       + label + '\"]';
 },
 /**
  * FSM.header:
  */
 FSM_header: function() {
  return 'digraph {\n';
 },
 /**
  * FSM.footer:
  */
 FSM_footer: function() {
  return '}\n';
 },
};
```

We note again that the implementation of most functions is quite straightforward. This is mainly due to the fact that these functions are *pure functions* in the sense that all relevant data is provided as parameters.

3 The OYAML Object Model Notation

OYAML is a sub-language of YAML for object (example) modeling. It is a light-weight, human-understandable and machine-readable textual notation. Like YAML files, OYAML files have ".yaml" as extension. Before we delve into OYAML we review basic notions of YAML.

3.1 YAML in a Nutshell

YAML [9] is a human readable data serialization language. It is broadly used for programming related tasks such as configuration files, internet messaging and object persistence.

YAML organizes data into different levels. It marks levels of data by indentation: data at the same level are aligned to the left with the same indentation. Spaces are to be used instead of tabs for indentation. One can use any number of spaces to indicate one level of indentation.

YAML supports the following data structures:

- Scalars
- Mappings
- Sequences

Scalar values (scalars in short) are the most basic and indivisible data type. Here are examples of scalar types and associated values:

```
String: Paul, Cat
Boolean: True, False
Integer: 5, 8
Floating Point: 3.14159, 314159e-05
```

A *mapping* is a set of key: value pairs. A colon and space (":␣") is used to separate the key from the value, with each key: value pair starting on its own line. For example:

```
lastName: Smith
firstName: Paul
age: 18
isMarried: False
```

A *sequence* is a list of data items. We use a dash followed by a space ("-␣") to indicate each entry/element in the sequence. For example:

```
- Cat
- Dog
- Goldfish
```

Sequences and mappings can be nested. Note the use of "#" to start single line comments in (O)YAML. For example:

```
# a mapping with two key value pairs
languages:
   # the first value is a sequence
   - YAML
   - Ruby
   - Perl
   - Python
websites:
   # the second value is a mapping
   YAML: yaml.org
   Ruby: ruby-lang.org
   Python: python.org
   Perl: use.perl.org
```

3.2 From YAML to OYAML

OYAML is a sub-language of YAML for object (example) modeling. The top level data structure in an OYAML object model is a sequence. Each element of the top level sequence represents an object. An object is represented by a singleton mapping "key:␣value", where:

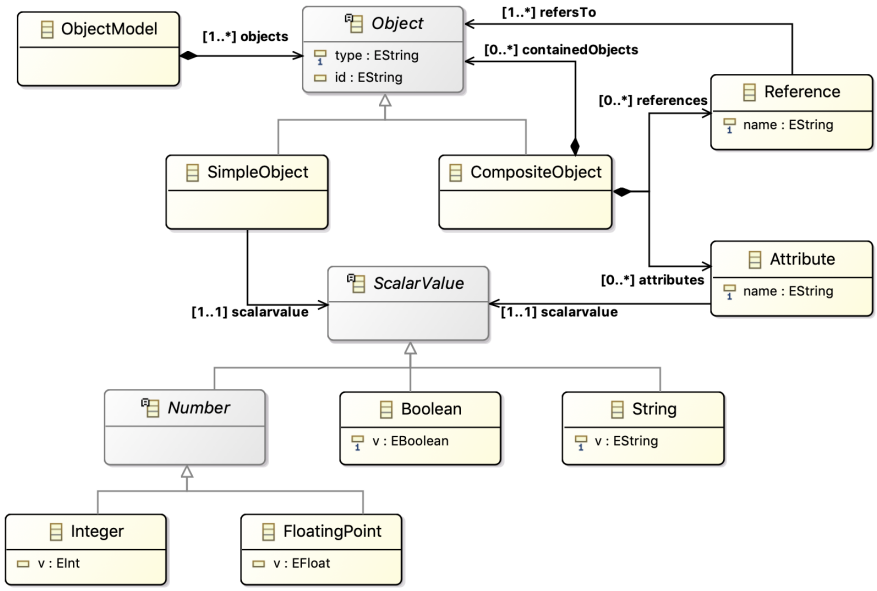

Figure 3 – Abstract Syntax of OYAML given as a Metamodel

- "key" indicates the type (and optionally the id) of the object, when the id is present, type and id are separated by a space, namely "type␣id" (Note: types must start with upper case letter.);
- and "value" gives the content of the object.

There are two kinds of objects in OYAML:

- simple objects: the value of a simple object is a scalar;
- composite object: the value of a composite object is a sequence. Elements of the sequence can be either:
  - an attribute: represented by attributeName:␣attributeValue, where attributeValue is a scalar value
  - a reference: represented by referenceName␣>:␣referenceValue, where referenceValue is a comma-separated list of object ids.
    Note: attribute and reference names must start with a lower case letter.
  - a contained object

For an example of an object model written in OYAML the reader is referred to the one given in section 2.

We can define the abstract syntax of OYAML by the metamodel given in Figure 3.

4 Functional Decomposition

Notation: We use angle brackets < and > surrounding comma-delimited elements to denote sequences.

In mathematics *functional decomposition* [4] is a technique for decomposing a function f into simpler functions such that the initial function f can be reconstructed from those simpler functions using function composition. More precisely if f is a multivariate function

$f(x_1, \ldots, x_n)$, functional decomposition amounts to identifying a set of multivariate functions $g_i(x_1, \ldots, x_n)$ and a function F such that

$$f(x_1, \ldots, x_n) = F(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$$

We call F the *decomposition function* throughout this paper. One key advantage of functional decomposition is that in many cases the functions g_i's are simpler than the initial function f by depending for instance on fewer variables. In these cases functional decomposition may contribute to reducing the complexity of the functional description.

In the context of this paper we do not deal with multivariate functions but rather with functions defined over object models. We start by defining the basic notions of object model and typed function.

Definition 1 (Object Model, Object, Reference) *An* object model *m consists of a set of* objects*, each having a type. An object in m may refer to an ordered sequence of other objects via a* reference*. If object o refers to the sequence of objects $< o_1, ..., o_k >$ via reference r, we write this as: $o.r =< o_1, ..., o_k >$.*

Remarks: in the equality $o.r =< o_1, ..., o_k >$ we omit the mention of m because m will generally be clear from the context. We only consider the case of ordered references since for unordered references we can simply ignore the order. Also note that we do not distinguish between attributes and references since attributes can be thought of as references whose target type is a scalar type.

Definition 2 (Successors, Predecessors) *Let r be a reference and o be an object. If $o.r =< o_1, ..., o_k >$, we call each o_i an r-successor of o and o an r-predecessor of each o_i.*

Definition 3 (Centered Object Model) *A* centered object model - centered model *for short - is a pair $M = (m, o)$ where m is an object model and o an object in m. We call o the* center *of M.*

Definition 4 (t-model and t-function) *A t-model is a centered model whose center is of type t. A t-function is a function whose domain only consists of t-models. We write a t-function f as $t.f$ and the image of f for t-model (m, o) as $t.f(m, o)$*

Definition 5 (Typed Function) *A typed function is a t-function for some type t.*

We are now ready to define the decomposition of typed functions.

Definition 6 (Decomposition of Typed Function) *A set of typed functions $\{t.g_1, \ldots, t.g_k\}$ is a* decomposition *of typed function $t.f$ if there exists a function F such that, for all t-models (m, o) in the domain of $t.f$, (m, o) is also in the domain of every $t.g_i$ and*

$$t.f(m, o) = F(t.g_1(m, o), \ldots, t.g_k(m, o))$$

We say that F is the decomposition function *for the decomposition of $t.f$. We shall assume that if the decomposition is empty then $t.f = F()$ is a* constant function*.*

For decomposing a typed function the following special functions come in handy:

Definition 7 (Reference Function) *The reference function for type t and reference r is defined by: $t.f(m, o) = o.r$ for any t-model (m, o).*

Definition 8 (Center Function) *For any type t the function $t.center$ returns the center of the argument t-model.*

Definition 9 (Forward Function) *A forward function for type t, reference r and t'-function $t'.g$, denoted by $t.(r \to t'.g)$, is defined by*

$$t.(r \to t'.g)(m, o) =< t'.g(m, o_1), \ldots, t'.g(m, o_k) >$$

where $< o_1, \ldots, o_k >$ is the sequence of those r-successors of o that are of type t'.

Definition 10 (Reverse Function) *A reverse function for type t, reference r and t'-function $t'.g$, denoted by $t.(r \leftarrow t'.g)$, is defined by*

$$t.(r \leftarrow t'.g)(m, o) = \{t'.g(m, o_1), \ldots, t'.g(m, o_k)\}$$

where $\{o_1, \ldots, o_k\}$ is the set of those r-predecessors of o that are of type t'.

We can represent the decomposition of a typed function conveniently by a graph.

Definition 11 (Decomposition Graph) *A* decomposition graph for a typed function $t.f$ *is a directed graph $D_{t.f}$ defined as follows:*

- *The vertices of $D_{t.f}$ are typed functions.*
- *There are three types of edges:*
  - *Local edges, denoted by $(t.g_1, t.g_2)$*
  - *Forward edges labeled by a reference r, denoted by $(t.g_1, r, t'.g_2)$*
  - *Reverse edges labeled by a reference r, denoted by $(t.g_1, \sim r, t'.g_2)$*
- *There is a mapping d from each edge e to a typed function d_e defined as follows:*
  - *For a local edge $e = (t.g_1, t.g_2)$: $d_e = t.g_2$*
  - *For a forward edge $e = (t.g_1, r, t'.g_2)$: $d_e = t.(r \to t'.g_2)$*
  - *For a reverse edge $e = (t.g_1, \sim r, t'.g_2)$: $d_e = t.(r \leftarrow t'.g_2)$*

There are two constraints that $D_{t.f}$ must satisfy:

1. *Typed function $t.f$ is a root of this digraph, i.e., there is a path from $t.f$ to every vertex in this graph.*
2. *For every node $t'.g$ with at least one outgoing edge: the set of functions $\{d_e : e$ is an outgoing edge of $t'.g\}$ is a decomposition for $t'.g$.*

Definition 12 (Complete Decomposition Graph) *We say that the decomposition graph $D_{t.f}$ is* complete *if every sink node in $D_{t.f}$ is either a constant function, the center function, or a reference function.*

We present the complete decomposition graph corresponding to the transformation `FSM.fsm2dot` defined in Section 2 in Figure 4. Note that reverse edges have their label preceded by '$\sim$'

5 Generalized Typed Decomposition Graphs

The basic setup for typed functions we have just described is not always applicable. Suppose a function $t.f(m, o)$ wants to call a function g on (m, o_i) for each r-successor o_i of o. If all o_i's have the same type t' the singleton set $\{t.(r \to t'.g)\}$ is a straightforward decomposition of $t.f$.

In the current rather relaxed definition of object models an object may refer to objects of different types. In this case there is no obvious way to decompose $t.f$. To deal with such a case we introduce a new type Object, assuming this to be a supertype of all existing types. The set $\{t.(r \to Object.g)\}$ can then be viewed as a decomposition of $t.f$, assuming that for each type t_i of an r-successor of o there is a redefinition of function g as typed function $t_i.g$. Note that this in fact introduces polymorphism to functional decomposition.

Adding polymorphism leads to generalizing the notion of decomposition graph.

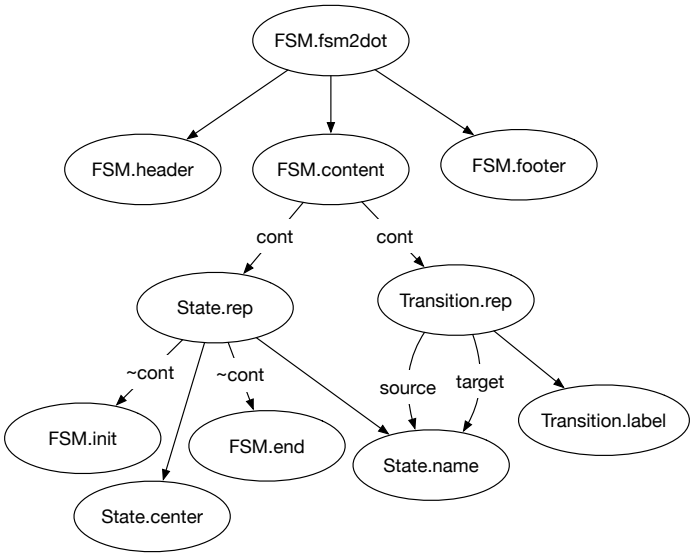

Figure 4 – Decomposition Graph of Transformation `FSM.fsm2dot`

Definition 13 (Generalized Decomposition Graph) *A generalized decomposition graph for a typed function $t.f$ is a decomposition graph that may include Object-functions and an additional edge type called* redefinition edge*: a redefinition edge links a function Object.g to a function $t'.g$ with the meaning that $t'.g$ redefines Object.g on t'-models. There is an additional constraint that only Object-functions may have outgoing redefinition edges. Furthermore if Object.g has at least one outgoing redefinition edge, then all outgoing edges must be redefinition edges.*

We may view decomposition graphs as a special case of generalized decomposition graphs that happen not to include Object-functions.

Suppose that we have a generalized decomposition graph for a t-function $t.f$. We can compute $t.f$ using the following procedure:

```
compute(t.f, m, o): // (m,o) is a t-model
  if t.f has no outgoing edges: // i.e., it is a sink node in decomposition graph
    if t.f is a constant function:
      return F() // constant value defined by decomposition function F of t.f
    else if t.f is a center function:
      return o
    else: // f must be a reference function for some reference r
      return o.r
  else: // at least one outgoing edge
    if t.f has outgoing redefinition edges:
      // all outgoing edges must then be redefinition edges and t = Object
      let t' be the type of o
      // there must be a redefinition edge from Object.f to t'.f
      return compute(t'.f, m, o)
    else: // local, forward or reverse outgoing edges
      for outgoing edges e1,...,ek of t.f:
        if ei is local edge (t.f,t.g):
          vi = compute(t.g, m, o)
        else if ei is forward edge (t.f, r, t'.g):
          vi = <compute(t'.g, m, oi): i = 1,...,k>
          // where <o1,...,ok> is sequence of r-successors of o of type t'
        else: // ei is reverse edge (t.f, ~r, t'.g)
          vi = {compute(g, m, oi): i = 1,...,k}
          // where {o1,...,ok} is set of r-predecessors of o of type t'
      return F(v1,...,vk)// F is decomposition function for t.f
```

The above `compute` procedure reduces the computation of typed function $t.f$ to that of computing the decomposition functions associated with the nodes in the decomposition

graph. This procedure is called by our tool when executing a transformation. Of course some non-trivial "glue" code is involved to make sure that this works for different target languages but this is not visible to the user.

6 The FUDOMO Language

To keep the overall approach lightweight in terms of tooling and involved languages, we shall introduce a simple textual notation for the decomposition graph. The notation is defined by the following BNF grammar:

```
<transformation> ::= <decomposition> | <transformation> <decomposition>
<decomposition> ::= <typedFunction>":" <links>
<links> ::=  " " | <link>  {"," <link>}
<link> ::=  <localLink> | <forwardLink> | <reverseLink>
<localLink> ::= <untypedfunction>
<forwardLink> ::= <reference> "->" <typedfunction>
<reverseLink> ::=  <reference> "<-" <typedfunction>
<typedFunction> ::= <type> "." <untypedfunction>
```

We note that `<reference>`, `<untypedfunction>` and `<type>` are all represented by string identifiers; the corresponding productions are omitted in the above grammar. We also note that redefinition edges do not occur in these decompositions. Indeed we will assume that redefinition edges are implicitly defined as follows: if Object.g occurs on the right-hand side of the decomposition it is redefined by all typed functions of the form $t.g$ on the left-hand side of decompositions for some type t.

Let us illustrate the use of the grammar with an excerpt from the example from section 2:

```
FSM.fsm2dot: header, content, footer
FSM.content: cont -> State.rep, cont -> Transition.rep
State.rep: name, cont <- FSM.init, cont <- FSM.end, center
```

The first decomposition contains 3 local links. The second decomposition contains two forward links, and the third decomposition contains two reverse links and a local link.

7 Case Study: the presi2beamer Transformation

With the help of the FSM example we illustrated in section 2 some concepts and features of our approach. We need a more complete and realistic example to better bring out the advantages of using our approach but also to illustrate an important feature that has not been used in the FSM example, namely redefinition edges. The transformation we now describe takes as input an object model describing a presentation (in fact a lecture) and the output is LaTeX code that can be used to render the content as a PDF file suitable for presentations. The generated LaTeX code is based on the use of the beamer class [2], a LaTeX class specifically designed for creating presentations.

Here is an excerpt of an example input model written in OYAML:

```
- Lesson:
    - course: Programming 1
    - number: 6
    - teacher: John Doe
- Section: Files
- Slide:
    - title: Text
    - Bullets: |
        The very first code snippet we saw in
        this course is the following (from lesson 1):
    - Code: |
        n = int(input('Please enter a number: '))
        print(-n)
    - Bullets: |
```

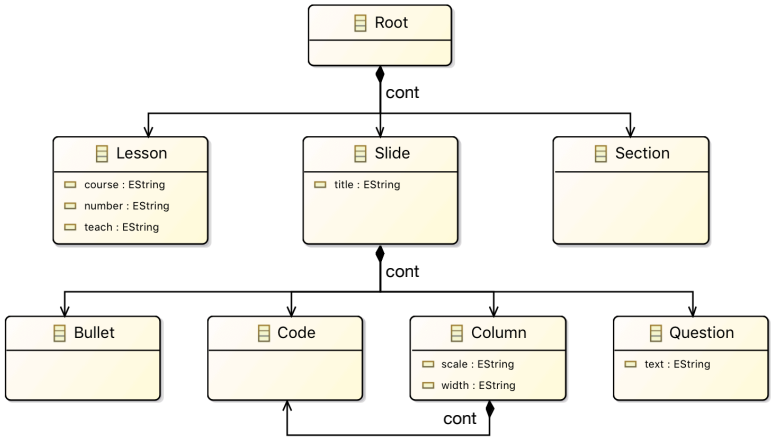

Figure 5 – Metamodel Inferred from the Example Lecture Presentation

```
        The program uses the \lstinline{input}
        function which reads a line of \alert{text} from the
        keyboard. Each line is terminated by the
        "non-printable" newline character,
        denoted {\tt $\backslash n$}
# comment: some slides omitted
- Slide:
    - title: Variable Naming - Advice
    - Question:
        - text:|
            Which of the following two code snippets
            is easier to understand, and why?
    - Column:
        - width: .3\textwidth
        - scale: small
        - Code: |
            x1 = 3.14
            x2 = 13.1
            c = x1*(x2**2)
    - Column:
        - width: .4\textwidth
        - scale: small
        - Code: |
            pi = 3.14
            diameter = 13.1
            area = pi*(diameter**2)
```

The above object model is a sequence of 4 objects of types `Lesson`, `Section`, `Slide` and `Slide` again. One of the four objects, the `Section` object, is a simple object in the sense that its value is a single string (a so-called scalar type in YAML). The other three objects are composite, meaning their value is again a sequence of elements. The Lesson object has three attributes named course, number, and teacher, telling us that this is the sixth lesson that was taught by John Doe in the context of the "Programming 1" course. The corresponding metamodel inferred by the FUDOMO tool, presented in Figure. 5 in a graphical form, summarizes the structure of lecture presentations.

As we already said, the `Section` object is a simple object. It tells us that the active (or current) section is called "Files". The following `Slide` objects are again composite objects, their value being a sequence of elements. As was the case for the `Lesson` object, the sequence for each `Slide` starts with attributes, in this case a single attribute named `title`, telling us the title of the slide. Unlike the `Lesson` object the `Slide` object contains other objects, namely, those elements that start with an upper-case identifier denoting a type name. In this case the first slide contains a sequence of three objects of types `Bullets`, `Code`, and `Bullets`

again. Type `Bullets` indicates a bulleted list and type `Code` indicates a code snippet. We notice that these three objects are all simple objects because their values are scalar (strings in this case). Note that the vertical bar "|" introduces a multi-line string. In such a string no special escape characters are needed for special characters because of the indentation. The second `Slide` contains a question and a pair of columns, each containing a code snippet.

The following is an excerpt of the decomposition graph written in the FUDOMO language introduced in the previous section.

```
Root.presi2beamer:
  header, body, footer
Root.header:
  cont -> Lesson.course, cont -> Lesson.number,
  cont -> Lesson.teacher
Root.body:
  cont -> Slide.f
Root.footer:
Slide.f:
  header, body, footer
Slide.header:
  title, hasCode, activeSection
Slide.hasCode:
  cont -> Bullets.hasCode, cont -> Code.center,
  cont -> Verbatim.center, cont -> Column.hasCode
Bullets.hasCode: val
Slide.activeSection:
  cont <- Root.cont
Slide.body:
  cont -> Object.f
Slide.footer:
Column.f:
  cont -> Object.f, isFirst
Column.isFirst:
  cont <- Slide.firstCol, center
Column.hasCode:
  cont -> Code.center
Slide.firstCol:
  cont -> Column.center
Code.f: val
Bullets.f: val
Question.f: text
```

We will now discuss the typed functions that intervene in the transformation. The function `Root.presi2beamer` represents the main transformation function. It simply returns the concatenation of a header, body and footer, each represented by a typed function.

The `Root.header` function produces LaTeX code containing the preamble. Part of the preamble is static, not depending on the object model. The part that depends on the presentation model is given in this example by the following lines of LaTeX-code:

```
\title{Programming 1}
\subtitle{Lesson 6}
\author{John Doe}
```

We notice that the LaTeX code is a mixture of static parts (latex commands) and dynamic parts (the actual data), in this case depending on the course, number and teacher attributes.

The `Root.body` function accounts for the bulk of the generated LaTeX code: it computes the transformed image of each slide via the `Slide.f` function (returning a string) and concatenates

the resulting strings of each slide (in the order the slides have in the presentation model). The output of each slide, computed by `Slide.f` is composed of a header followed by the body and the footer, in that order.

As an example here is the header of the output for the first slide in the above object model:

```
\begin{frame}[fragile]
\frametitle{Files}
\framesubtitle{Text}
```

Although this output is quite short, it is not so trivial to compute. First we notice that this frame is "fragile". In Beamer a frame needs to be declared as fragile if it contains code (in lstlisting environment) or verbatim text. This is the purpose of the `hasCode` function: it returns true if a slide contains code or verbatim text. The frame title is the name of the activeSection, which is computed by the function of the same name in the decomposition of `Slide.header`. Here is code snippet that computes the header of the slide. For presi2beamer we use as target language Javascript so this snippet is written in Javascript. We note that the code is fairly straightforward because the supplied arguments provide the needed information.

```
Slide_header: function(title, hasCode, activeSection, label) {
 let res = '';
 if (hasCode) {
  res += '\\begin{frame}[fragile]\n';
 } else {
  res += '\\begin{frame}\n';
 }
 if (activeSection) {
  res += `\\frametitle{${activeSection}}\n`;
 }
 if (title) {
  res += `\\framesubtitle{${title}}\n`;
 }
 if (label) {
  res += `\\label{${label}}\n`;
 }
 return res;
}
```

Next we consider the `Slide.hasCode` function. This function returns true if a bulleted list contained in the slide contains (inline) code or if the slide contains a `Code` object or `Verbatim` object. Therefore we decompose `Slide.hasCode` using four forward edges labeled by reference `cont` with target functions `Bullets.hasCode`, `Code.center`, `Verbatim.center` and `Column.hasCode`. Recall that the typed function `center` - defined for any type - returns the center of a centered model. Here is the Javascript code for computing `Slide.hasCode`. Again it is fairly simple since we have deferred the main complexity to the functions in the decomposition, whose values are supplied by the parameters:

```
 Slide_hasCode: function(cont_Bullets_hasCode, cont_Code_center,
                  cont_Verbatim_center, cont_Column_hasCode) {
  let anyBulletHasCode = false;
  for (const bulletHasCode of cont_Bullets_hasCode) {
   if (bulletHasCode) {
    anyBulletHasCode = true;
   }
  }
  let anyColumnHasCode = false;
  for (const columnHasCode of cont_Column_hasCode) {
```

```
    if (columnHasCode) {
      anyColumnHasCode = true;
    }
  }
  return anyBulletHasCode
       || anyColumnHasCode
       || cont_Code_center.length > 0
       || cont_Verbatim_center.length > 0;
}
```

The function `Bullets.hasCode` essentially does a string search for `||` (signaling the presence of inline code) in its value string.

The `Slide.activeSection` function computes the active section of this slide. It decomposes into function `Root.cont` via a reverse edge labeled by reference `cont`. We note that there is a single `Root` object containing this slide so that the set of its `cont`-predecessors is a set having the `Root` object as single element. From `Root.cont` we can compute in a simple for-loop the active section: it is the last object in this sequence that precedes this `Slide` object.

The `Slide.body` function computes the body of the frame. It is obtained by computing the output for the contained objects using typed function `Object.f`, which is redefined (via implicit redefinition edges) by typed functions `Column.f`, `Code.f`, `Bullets.f` and `Question.f`.

The slide footer, computed by `Slide.footer`, has an empty decomposition since it is the constant string `\end{frame}`.

`Column` objects always come in pairs (as can be seen in the second Slide in our example). The first and second column in a pair have different headers and footers (not explicitly represented here). This explains the presence of typed function `Column.isFirst` in the decomposition; `Column.isFirst` returns true if it is the first element in the pair. To compute this function, we need to inspect the `Column` objects contained in the parent slide, explaining the reverse edge to the `Slide.firstCol` function.

Functions `Code.f` and `Bullets.f` compute the output for code snippets and bulleted lists. They only depend on their value string. Note that `Code.val` and `Bullets.val` are both reference functions that do not need to be decomposed further.

Last but not least function `Root.footer` returns the constant string `\end{document}` and thus has an empty decomposition.

8 FUDOMO in Action

Figure 6 presents the FUDOMO process. Two roles are involved in the execution of the process: (1) a domain expert who knows the application domain but does not necessarily have a technical background in (meta-)modelling and programming; (2) an engineer capable of working with a target language (a GPL such as Python or JavaScript or a formal language).

A role is attached to an activity to indicate who performs the action. If no role is attached, this means this activity is performed automatically by the FUDUMO tool.

The domain expert creates one or more examples in OYAML (activity 1). A metamodel can be inferred from the set of examples automatically by the FUDOMO tool (activity 2). New examples can be further provided by the domain expert (activity 1). Two options are available in the presence of new examples. One can either check the conformance of the new examples against the previously inferred metamodel (activity 3). In case an example does not conform to the metamodel, conflicts are reported to the domain expert, who then has the possibility to revise the example and make it conformant (activity 1). Alternatively, one can also extend the set of examples with the new example and infer a new metamodel from the extended set of examples (activity 2).

Once the metamodel reaches a stable stage, the domain expert can proceed with the model-to-text transformation specification. First (s)he will define the functional decomposition of

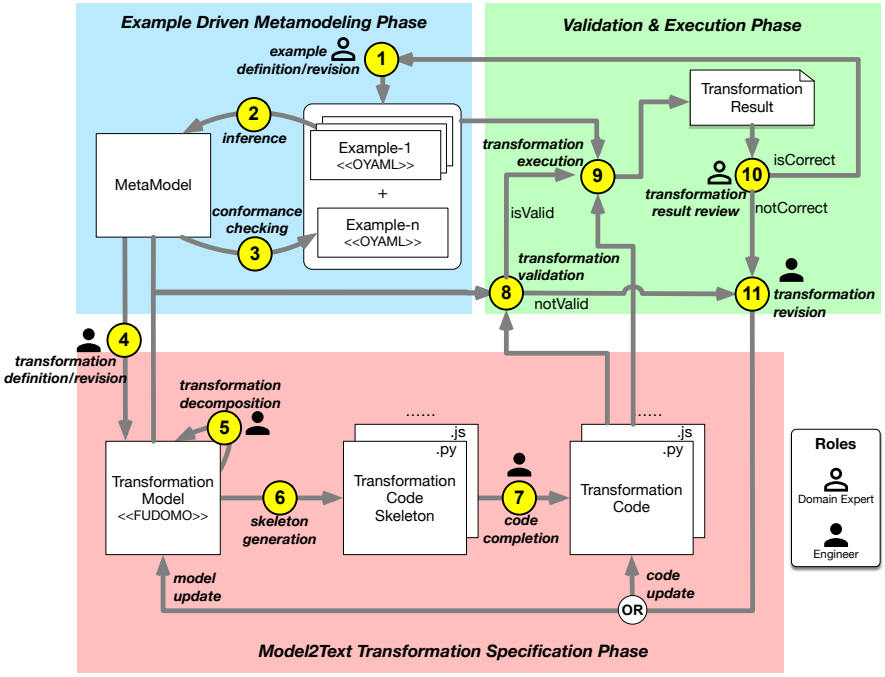

Figure 6 – FUDOMO Process

the transformation function in a FUDOMO model (activities 4 and 5). From the FUDOMO model, the tool can automatically generate implementation code skeletons in a selected target language (activity 6), such as Python and Javascript. An engineer capable of working with the selected target language will fill in the missing code implementing the computation in the generated skeleton (activity 7).

After completing the specification, the transformation will be validated (activity 8). The validation of a transformation entails two checks: firstly, the types and elements referred to in the transformation shall all be defined accordingly in the metamodel; secondly, the transformation code should be in sync with the transformation model, namely there is a function corresponds to a decomposition, and one parameter corresponds to a decomposition edge. If validation of the transformation fails, the transformation has to be revised (activity 11). A valid transformation can be executed (activity 9) on examples to produce transformation results, which are in text format. The domain expert can review the transformation results. If the domain expert judges the transformation result to be correct (activity 10), the domain expert may define more examples or revise existing ones (activity 1) to be transformed. Otherwise, if the result is not correct, revision of the transformation specification is needed (activity 11).

9 Discussion and Related Work

The purpose of this section is two-fold: first we review and evaluate our approach. Second we place the approach in the context of related work.

The declared main goal for developing our approach is the drive to make modeling more accessible. Let us elaborate on the main points for supporting this claim that ware already mentioned in the introduction:

- Our approach is example-driven rather than metamodel-driven: the user starts out

by creating concrete examples from which s(h)e can then infer a metamodel. That example-driven approaches make modeling more accessible has already been amply documented by research works in example driven modeling (see below).

A peculiarity of our approach is the use of a new human-readable textual notation OYAML for writing object models. The reason for introducing a new notation rather than using an existing one will be discussed in more detail below when review existing notations. The small size of the OYAML language (judging by the size of its metamodel given in Figure 3), should make it easy to learn. The compatibility with YAML allows to reuse tool support for YAML such as editors with syntax-highlighting and parsers. Finally the availability of YAML libraries for a multitude of languages [9] reduces the effort to add support in our tool for different target languages.

- The idea of basing a transformation language directly on functional decomposition in the way we do is novel as far as we know. Modeling the decomposition graph should be facilitated by the small size of the FUDOMO language (about 10 productions) and the clear separation of concerns, relieving the user from having to worry about the details of the transformation and instead focus on the data dependencies of the computation.

  The separation of the decomposition model and the computation of the decomposition function, with the latter one being expressed by pure functions, makes our framework extendible to a multitude of target languages. This should allow many users to reuse a GPL they already know, rather than having to learn a special purpose model transformation language.

Let us now compare our work with related approaches. We start by placing our approach in the context of research on example driven modeling. The work in [14] is high-level in the sense that the key ingredients of an example driven approach are discussed without advocating a particular implementation of these ideas. Two main activities are identified in the context of example driven modeling: *abstraction inference* for synthesizing abstractions from examples and *example derivation* for generating examples from abstractions. Our approach uses only abstraction inference: metamodel concepts and features are extracted from the concrete object models. The authors of [22] propose a concrete approach for developing metamodels based on an iterative abstraction inference process. Compared to our approach they provide a richer language for expressing metamodels and also foresee for the user to refactor metamodels. Unlike our approach transformations are not integrated in their approach but the metamodels can be compiled into implementation models in existing modeling frameworks such as EMF; traditional languages for model transformations such as ATL can then be used to write model transformations. The work in [13] presents a language named Clafer that can be used to represent both examples and abstractions. No particular transformation approach is presented. The main scope is domain analysis and requirements elicitation.

The OYAML object model notation that we introduce in this paper can be compared to the Human-Usable Textual Notation (HUTN) [8]. HUTN is an OMG standard for encoding MOF-based models. An implementation of HUTN has been done within the Epsilon modeling framework [5]. The reason for us basing our object modeling notation on YAML rather than HUTN is the availability of libraries for YAML for many GPLs, which greatly facilitates implementing parser and syntax highlighting in our tool for different target languages. Furthermore not being tied to a particular modeling framework allowed us to implement our tool on top of a web-based platform, not requiring any installation by the user and thus keeping tooling fairly lightweight.

Our approach is strongly inspired by that proposed by [19]: the authors of that paper propose a modeling language named EP allowing the specification of both structural and behavioral aspects of a system. The main concepts in the EP-language are events and properties. Query properties are similar to typed functions. The property graphs used to describe a functional decomposition of query properties are similar to our decomposition graphs. The main contributions of the present paper with respect to [19] are: on one side a more general description of the functional decomposition technique that only relies on a

rather general notion of object model, without relying on a particular modeling language, and on the other side the introduction of polymorphism (via redefinition edges) in the functional decomposition. Another difference is the use of a fixed target language (OCL) in [19] for implementing the code snippet while our approach allows the free choice of a target language.

An essential component of functional decomposition is the underlying navigation. The idea of separating navigation from computation is not new: for instance the Visitor Design pattern [18] is based on separating navigation (called traversal) from computation. Computation is encapsulated in visitor classes that are separate from the classes defining the types used in the object models. The Visitor pattern has a wider scope than our method since the operations specified in the Visitor classes do not need to be side-effect free, that is, they can modify objects. Our approach provides a more light-weight separation of navigation and computation: the "operations" are encapsulated into local functions. This is possible because we compute side-effect free functions. Another difference is the fact that we provide a more abstract description of the navigation in terms of the decomposition graph. This approach is declarative in the sense that a number of traversals can correspond to the same decomposition graph. We notice that a more flexible version of the Visitor pattern called the Guide pattern was proposed by [17]: this pattern allows to use a variety of navigation schemes.

Another example of an approach based on separating navigation (or traversal) from computation is adaptive programming [20]. The motivation behind adaptive programming is not so much complexity reduction but rather adaptiveness to changes in a class structure: by specifying traversals in terms of propagation patterns which can be satisfied by different class structures adaptive programs become less prone to modification when the underlying class structure changes. Computation is focused on essential calculations rather than those that simply pass on information.

The pure nature of the decomposition functions mentioned above is related to coupling between typed functions: each typed function only "talks" to typed functions whose argument (centered model) is an immediate successor or predecessor of the argument (centered model) of this function. This idea of objects only talking to their neighbors has been pioneered earlier by Ian Holland and is known as the Law of Demeter [21]. The Law of Demeter is a design principle for designing object-oriented programs: it promotes loose coupling by having each "unit" talk only to units directly related to it. We can view our approach as following the Law of Demeter when viewing the typed functions in the functional decomposition as units.

We close this section by comparing our example applications to existing approaches to model-to-text transformations. There are two major approaches to model-to-text transformations [16]: programmatically or through special model-to-text transformation languages. The first approach relies on writing a program (e.g., in Java) that queries the model using a model API. This can be done for instance using Java as a programming language and an EMF-based model API [23]. The advantage of this approach is that no additional programming skills are necessary but there are also some disadvantages: (i) static and dynamic parts (in the output) are intermingled; (ii) the output structure is difficult to grasp since it is embedded in code; (iii) large amounts of code are needed because there is no declarative query language for accessing models; (iv) code cannot be easily reused.

To remediate these disadvantages, DSLs have been developed for specifying model-to-text transformations. We will refer to these DSLs as M2T transformation languages. Examples of such languages are: Acceleo [1] (based on the MOF standard MOFM2T [6]), XSLT [11], and Xtend [12], to name just a few. These languages are typically template-based: they represent static and dynamic parts differently (for the latter ones they use meta-markers). The code for generating the dynamic parts is embedded in the static parts, thus making the structure of the output easier to comprehend. Less code is needed because declarative query languages are used to query models (usually OCL [7]). Because functionalities for importing models and serializing output are part of most M2T languages, code reuse is simplified.

We now review our approach to model-to-text transformations according the criteria we just considered: the static parts of the generated text correspond to constant functions. These

are fairly easily detectable because they correspond to leaves of the decomposition graph. Regarding the second criteria the output structure can be comprehended in a hierarchical manner by defining corresponding functions. For instance in the presi2beamer transformation the first decomposition tells us that the output is made up of header, body and footer. Each of these can be further explored to understand the structure of the output. As for the third criteria, the presence of a declarative query language: we do not require such a language because the necessary queries are done in the glue (or auxiliary) code that is generated once per target language. Note that this glue code is essentially a transcription of the "compute" procedure (given in section 5) into the target language. Finally this glue code represents excellent code reuse since it relieves us from writing intricate queries.

10 Conclusion

In this paper we present a modeling framework that combines an example-driven approach for modeling with a novel model transformation approach based on functional decomposition. The learning effort required for using the framework can be assessed in terms of the involved languages: OYAML, a human-readable notation for expressing object models, FUDOMO, a textual language for expressing functional decompositions of transformation functions, and a GPL for implementing the pure functions expressing the detailed computations. Both OYAML and FUDOMO are small languages, judging by their expressive power. By allowing the user to make use of a GPL to express the detailed behavior of transformation functions, without having to worry about navigation, our approach has the potential to make modeling indeed more accessible.

Our current support for example-driven modeling is rather rudimentary in the sense that constraints (even simple multiplicity constraints) cannot directly be included in the metamodel. Future work will explore to what extent constraints can be automatically inferred from the object models (so as to keep our modeling framework lightweight).

Because of the generality of the decomposition method, we expect that our technique to be applicable in other contexts. Another obvious next step would be to widen the application scope to general model transformations. In that context it would be important to use benchmark examples like the CD2RDBMS transformation [15] to compare our approach to existing approaches.

To assess the practicality of the approach the rather small examples treated in this paper are not sufficient. Because the tool is web-based and does not need any installation, we are counting on user feedback that should help in assessing the accessibility of the tool and in making the tool more relevant for practitioners.

References

[1] Acceleo. https://www.eclipse.org/acceleo/.

[2] beamer - A LaTeX class for producing presentations and slides. https://ctan.org/pkg/beamer.

[3] The DOT language. https://www.graphviz.org/doc/info/lang.html.

[4] Functional decomposition — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Functional_decomposition.

[5] Human Usable Textual Notation. https://www.eclipse.org/epsilon/doc/hutn/.

[6] MOFM2T. https://www.omg.org/spec/MOFM2T/About-MOFM2T/.

[7] Object Constraint Language. https://www.omg.org/spec/OCL/About-OCL/.

[8] Object management group. Human-Usable Textual Notation Specification (2004). http://www.omg.org/technology/documents/formal/hutn.htm.

[9] The Official YAML Web Site. https://yaml.org.

[10] The FUDOMO tool. https://fudomo.uni.lu.

[11] XSLT. https://www.w3.org/TR/xslt.

[12] XTend. https://www.eclipse.org/xtend.

[13] Michal Antkiewicz, Kacper Bak, Krzysztof Czarnecki, Zinovy Diskin, Dina Zayan, and Andrzej Wasowski. Example-driven modeling using clafer. In *MDEBE@ MoDELS*, volume 1104, pages 32–41, 2013.

[14] Kacper Bak, Dina Zayan, Krzysztof Czarnecki, Michal Antkiewicz, Zinovy Diskin, Andrzej Wasowski, and Derek Rayside. Example-driven modeling: model = abstractions + examples. In *ICSE '13*, pages 1273–1276. IEEE Computer Society, 2013.

[15] Jean Bézivin, Fabian Büttner, Martin Gogolla, Frédéric Jouault, Ivan Kurtev, and Arne Lindow. Model Transformations? Transformation Models! In *MoDELS*, pages 440–453. Springer, 2006.

[16] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. Model-driven software engineering in practice. *Synthesis Lectures on Software Engineering*, 1(1):1–182, 2012.

[17] Martin Bravenboer and Eelco Visser. Guiding visitors: Separating navigation from computation. *Utrecht University Repository*, 2001.

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[19] Pierre Kelsen. A declarative executable model for object-based systems based on functional decomposition. In *ICSOFT 2006*, pages 63–71, 2006.

[20] Karl Lieberherr. Adaptive Object-Oriented Software. The Demeter Method. *PWS Boston*, 1996.

[21] Karl J. Lieberherr and Ian M. Holland. Assuring good style for object-oriented programs. *IEEE Software*, 6(5):38–48, 1989.

[22] Jesús J. López-Fernández, Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. Example-driven meta-model development. *Software and Systems Modeling*, 14(4):1323–1347, 2015.

[23] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.