MT-Flow – An Environment for Workflow-Supported Model Transformations in MDA

Jernej Kovse and Theo Härder

Department of Computer Science
Kaiserslautern University of Technology
P.O. Box 3049, D-67653 Kaiserslautern, Germany
{kovse,haerder}@informatik.uni-kl.de

Abstract. Specification of systems in a software product line (product-line members) is often supported by domain-specific languages (DSLs) that provide powerful language abstractions for selecting the features of the desired system. In this paper, we show that efficient composition of system specifications (which, in our case, are expressed as models) is also possible using (i) a domain-specific workflow model that guides the composition and (ii) a set of domain-specific templates for model transformations. We illustrate the entire approach on a product line for versioning systems, define a metamodel for workflow models and postulate a measure for estimating the benefits of the proposed approach.

1 Problem Formulation

Generative software development approaches, extensively outlined by Czarnecki and Eisenecker [6], represent a key enabling technology for *software product lines*, which have recently gained tremendous attention both from research and industry. According to Clements and Northrop [4], a *software product line* is a set of software-intensive systems sharing a common, managed set of features that satisfy the needs of a particular market segment. However, each member from this set will still possess an amount of unique functionality. A common example is to have a software product line for systems that control the performance of truck engines. Such a product line is supported by Cummins Engine Inc. [4,5], the world's largest manufacturer of commercial diesel engines above 50 horsepower. Engine types manufactured by the company come with different characteristics: engines may range from 50 to more than 3500 horsepower, have 4 to 18 cylinders, and operate with a variety of fuel systems, air-handling systems, and sensors. The software (with typical size of 130 KSLOC) that controls the engine's performance to produce an optimum mix of power, fuel consumption and emissions, has to be fully optimized for the characteristics of the engine. In cases like this one, companies will try to take advantage of the commonalities among the products in the product line to avoid developing the software for each product from scratch. A possible way to achieve this is to employ a *generator* that, once desired properties of the product are described, takes advantage of *a shared code framework* and *generation templates* to automatically produce the software that corresponds to this product.

A. Persson and J. Stirna (Eds.): CAiSE 2004, LNCS 3084, pp. 160–174, 2004.

Customers that want to buy a software system embraced in a product line have to be given a possibility to clearly describe the system they want. This process is also called *ordering*, since a formal *specification* (*order*) that describes the configuration of features to be included in the system has to be devised. Ideally, this order will serve as an input to a generator that will automatically assemble the system. In the context of product line engineering, a lot of research has been done on *domain-specific languages* (*DSLs*), which can be used to support the ordering process - a program in a DSL is considered a formal specification for the system embraced in the product line. As mentioned by Czarnecki and Eisenecker [6], DSLs are highly specialized, problem-oriented languages as they restrain from using *language elements* (*abstractions*) that are irrelevant for the domain of the product line.

As a main advantage of DSLs we recognize small specification size: Since the abstractions they support are highly specialized for the domain of the product line, the desired system can be specified very briefly and the formal specification (the DSL program) can easily be analyzed and discussed among developers. However, DSLs also come with many drawbacks. First, a separate DSL with the associated parser for DSL programs has to be developed for each product line. In case a DSL is graphical, it will require a specialized editor for devising DSL programs (specifications). Finally, each DSL requires a *dedicated generator* that understands the semantics of DSL abstractions to map them onto the implementation. Approaches like Intentional Programming (IP) [23] employ metaprogramming techniques to support accelerated development of environments for editing, compiling (generating), and debugging domain-specific programs.

In this paper, we claim that, although widely recognized, DSLs are not the only convenient way for ordering products from a software product line. Our *MT-Flow (Model Transformation workFLOWs)* approach proposes that, instead of devising a product order in form of a DSL program, such an order can as well be expressed in a general-purpose specification language, such as Executable UML [12]. MT-Flow uses a *workflow model* to guide the user through a series of *system configuration steps*. On each of these steps, the user selects the features of the system he wants to order. The process is similar to configuring mass-customized products (e.g. cars, kitchen settings, or computers) in the case of non-software product lines.

At the end of each configuration step, MT-Flow *transforms* the current state of the model, so that the decisions the user has made can be immediately observed in a modeling tool. The user selects system features within a graphic interface, which is substantially different from DSL programming, where the user first has to learn the syntax and semantics of the DSL. Another common problem when using DSLs is assuring that a DSL program represents a semantically correct specification. This is difficult until the program is completed, since there may be parts of the program the user yet attempts to write, but at some intermediate programming stage the absence of these parts appears as a correctness violation. This is not possible in our approach, because the workflow model required by MT-Flow determines the paths in which the specification is allowed to evolve: The *flow of control* among the steps prevents an incompatible selection of features and thereby assures that the choices made in the individual steps will always lead to a semantically correct specification. MT-Flow is a generic approach, meaning

that its application is not limited to a single product line: The workflow model is *defined externally* and merely interpreted by MT-Flow (MT-Flow acts as a workflow engine). This means that various software product lines described using a general purpose modeling language based on the MOF Model [15] can be supported by simply exchanging the workflow model.

The rest of this paper is organized as follows. Sect. 2 will put the problem in the context of the OMG's Model Driven Architecture (MDA). A motivating example that illustrates the application of MT-Flow is given in Sect. 3. Sect. 4 defines a metamodel for MT-Flow's workflow models and illustrates the use of transformation templates. In Sect. 5, we define a measure to estimate the benefits of our approach. Sect. 6 gives an overview of related work. Finally, our results are summarized in Sect. 7, which also outlines some ideas for the future work.

2 Relation of the Problem to OMG MDA

OMG's *Model Driven Architecture* (MDA) [14] is an exciting approach to developing software systems: First, the *specification of a system* is expressed in form of a *model* in a selected *modeling language*. Afterwards, a *generator* maps this model to the *implementation of the system* on a particular execution platform.

As identified by Frankel [7], developers have four different options when choosing the modeling language for the specification. These options are best illustrated in terms of the OMG's four layer metadata architecture (Fig. 1).

- *Approach 1: Using unadorned (standardized) modeling languages*. In this case, the modeling language is an existing standardized OMG modeling language, such as UML or CWM.
- *Approach 2: Using lightweight language extensions*. Some modeling languages allow developers to define new language constructs *using the language itself*. The most prominent example of this approach are *UML profiles*, which are a built-in extensibility mechanism of the UML language. A UML profile will typically define many *stereotypes*, which are used by developers to *brand* elements in their models. In this way, an element attains supplementary semantics in addition to the semantics defined by its type in the UML Metamodel.

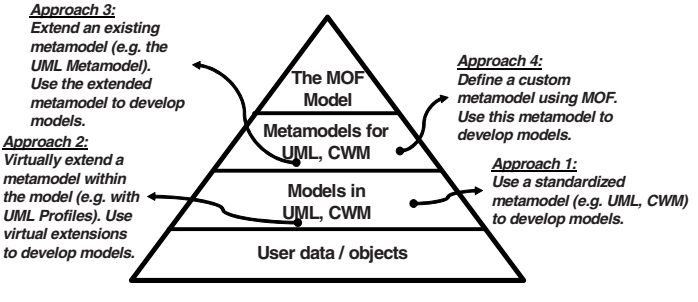


Fig. 1. Choosing the modeling language for the specification

- *Approach 3: Using heavyweight language extensions*. As an alternative to *Approach 2*, new language constructs can also be introduced by extending the metamodel of the modeling language. For example, the UML Metamodel can be enriched with additional types that we connect to the standardized metamodel types by means of generalization and associations.
- *Approach 4: Creating new modeling languages*. When applying *Approaches 2 and 3*, developers will sometimes feel burdened with the types defined by the metamodel of the language (e.g., the UML Metamodel), because the types may not be significant for their problem domain. The problem with *Approaches 2 and 3* is that in both cases, the extensions of the modeling language occur in an *additive* fashion only – the existing language constructs, i.e. types from the metamodel, cannot be modified or deleted. For such problem domains, developers can choose to define an entirely new modeling language from scratch. The metamodel for the new language will be defined in terms of the MOF Model, which is an OMG-standardized meta-metamodel used for defining different metamodels.

It is our observation that the new constructs defined in *Approaches 2*, *3*, and *4* actually represent the use of a domain-specific modeling language. Recent approaches like DSL-DIA [11] aim at easy development of custom generators that understand new modeling languages defined using *Approach 4*. Except for *Approach 2*, where the use of a UML profile is largely supported by UML modeling tools, *Approaches 3* and *4* require a special modeling environment. Many *Computer-Aided Method Engineering (CAME)* approaches [22], like MetaEdit+ [13], involve automatic production of problem-oriented modeling environments to be used for new modeling languages.

By MT-Flow, we show that *problem-oriented modeling* (which is supported by domain-specific modeling constructs in *Approaches 2*, *3*, and *4*) is equally possible in *Approach 1*, provided there exists a set of *domain-specific model transformation templates* (used to produce *concrete transformations*) and an associated *domain-specific workflow model* that guides their application. Since the specification is expressed using a widely accepted (standardized) general-purpose modeling language, such as UML, the problem of finding an appropriate model-based generator that will map the model to the implementation or a model-based virtual machine capable of executing the model is eliminated. For example, MC-2020 and MC-3020 [19] are available compilers capable of mapping Executable UML models to C++ and C code, while the design and implementation of a virtual machine for UML is discussed by Riehle et al. [20].

Concrete transformations produced by MT-Flow from transformation templates to transform the model can clearly be categorized as PIM-to-PIM transformations mentioned by the MDA specification document [14]. These transformations enhance or filter a model without needing any platform-dependent information. The design of a PIM-to-PIM transformation does not prove very economical in case a transformation will be used in the design of a single system only. Instead (and this is the goal of MT-Flow's transformation templates) we want to capture recurring model enhancement or filtering patterns that can be reused when modeling many systems. General *design patterns* (that do not introduce new semantics to the modeled system, but improve its *architecture*), such as those proposed by Gamma et al. [8] can be found even for very dislike systems. However, specialized patterns (these are the ones we are interested in) that sequentially

introduce new semantics to the model can be defined only for systems with a controlled set of common and variable features, i.e., when the systems constitute a product line.

In MT-Flow we choose *differential model exchange* supported by OMG XMI [16] for expressing concrete transformations. In this approach, also referred to by the specification document [16] as *model merging*, a difference between two models is seen as a transformation that takes the old model to the new model. The transformation is described by four tags used for expressing differences: `XMI.difference` (the parent element), `XMI.delete`, `XMI.add`, and `XMI.replace`. MT-Flow is not bound on using the UML as the modeling language, since differential model exchange is supported for any MOF-based metamodel [16].

3 Motivating Example: A Product Line for Versioning Systems

We choose a *product line for versioning systems* as a motivating example for MT-Flow. Why do we treat versioning systems as a product line? Each versioning system possesses a unique *information model* [2], i.e., type definitions for objects and relationships to be stored and versioned (e.g., as evident from the following example, an information model used for a project management application defines object types like *Task*, *ProjectOffer*, and *Employee*).

Next, we may want to refine the information model with the following definitions.

- Which object types support versioning? An object type that supports versioning provides a method *createSuccessor* (for deriving a successor to an object version) and a method *freeze* (for freezing an object version).
- Which relationship type ends are *floating*? Floating relationship ends implement the behavior needed for *versioned relationships*. In case a given *project offer*, *p*, possesses a reference to a *task*, *t*, and *t* is versioned, a floating relationship end at *t* contains a user-managed subset of all versions of *t*. This subset is called a *candidate version collection* and may have a *pinned version*, which is the default version to be returned in case the client does not want to select a version from the collection manually. Maintaining floating relationship ends implies performance impedance so the user may choose to omit this feature for some relationship types.
- How do object and version management operations like *new*, *copy*, *createSuccessor*, *freeze*, *checkout/checkin* propagate across the relationships? A detailed coverage of operation propagation is given by Rumbaugh [21].
- What are the *workspace types* and how do these relate to regular object types? A workspace acts as a container for regular objects. However, only one version of an object may be attached to a workspace at a time. For this reason, a workspace acts as a single-version view to objects stored in the versioning system.

Many commercial versioning systems are implemented *generically* – this means that a large part of their implementation does not need to change for each user-defined information model. For example, a generic system can define a single database table for storing relationships (of every possible relationship type) and a single table for storing versioning information (like information on a predecessor version for a given version). Both tables are present for every installed information model, while tables for object

types are specific for a given information model. A *fully generative approach* (like the one supported by MT-Flow), on the other hand, allows the user a fine-grained specification of desired features in an information model that will also contain user decisions on floating relationship ends, operation propagation, workspace type definitions, etc. In this case, each information model (i.e., the specification) can be mapped to a distinct, completely generated and thus optimized implementation of a versioning system. The following sections will illustrate how MT-Flow supports the development of an information model in Executable UML for a simple versioning system.

3.1 Step A: Define an Object Type

In this step, the user defines an object type. This includes the definition of a name for the object type and the attributes with the corresponding primitive data types. The interface shown by MT-Flow for this purpose is illustrated in Fig. 2. We use this interface to define the object type *Task*. MT-Flow will fill an existing transformation template with these definitions. As a result, a concrete transformation is expressed using XMI differential tags. The transformation adds a new UML class *Task* (see Fig. 2) with its attributes to the current model (we assume we begin with an empty model in each specification). In addition, an *objId* that identifies an object within the versioning system and a *globalId* that identifies a particular *object version* within the versioning system are added. *Step A* can be repeated many times to add additional object types. We repeat the step to add object types *ProjectOffer* and *Employee*. The final model obtained in this way is illustrated in Fig. 2.

3.2 Step B: Define Which Objects Are Versioned

In this step, the user specifies which object types support versioning. As an additional feature, he may choose to limit the allowed number of successors to a version. We carry out *Step B* twice to make *Task* and *ProjectOffer* versionable. Concrete transformations will add a *verID* (a unique identifier of a version within a version tree) to the classes, a reflexive association for linking a version to its successors, a *createSuccessor* operation, and a *freeze* operation. The semantics of these operations is expressed using the model elements of the *UML Action Semantics* package, which can have many possible

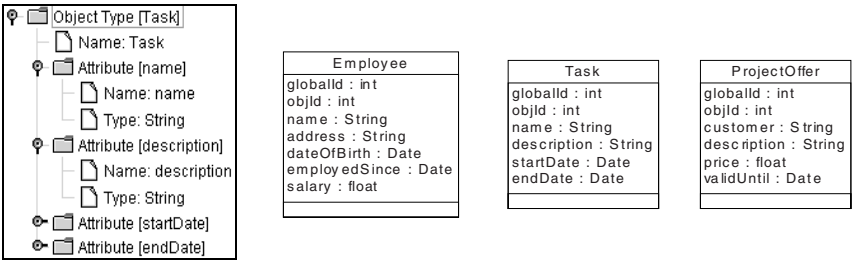
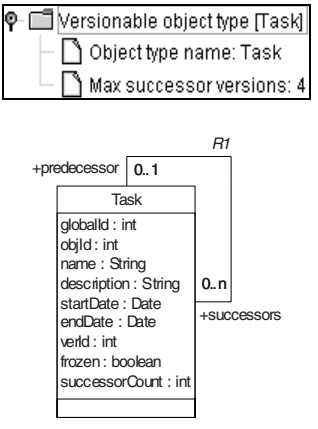


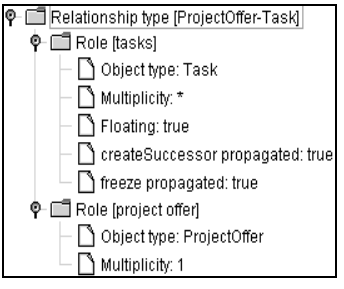
Fig. 2. Adding a new object type

syntax renderings in so-called *action languages*. For this reason, actions appear as text in diagrams. In the model, however, they are represented as instances of modeling constructs defined by the Action Semantics package, and can therefore be manipulated using XMI's differential elements. In Fig. 3, showing the result of making the class *Task* versionable, we use the Object Action Language (OAL), proposed by Mellor and




```
Task::createSuccessor
entry//
// check whether version is frozen
if (self.frozen == false) and (self.successorCount < 4)
    // create new instance of Task, set the ids, copy user-defined attributes
    create object instance newTask of Task;
    newTask.objId = self.objId; // should have the same objId
    newTask.verId = VerIdCounter.getNextVerId(self.objId); // get a new verId
    newTask.globalId = newTask.objId*10000 + newTask.verId;
    newTask.name = self.name;
    newTask.description = self.description;
    newTask.startDate = self.startDate;
    newTask.endDate = self.endDate;
    newTask.frozen = false;
    newTask.successorCount = 0; // Does not have any successors yet
    // relate the successor version to its predecessor
    relate newTask to self across R1.'successors';
    // increase the number of own successors
    self.successorCount = self.successorCount + 1;
else
    // generate error
end if;
```

Fig. 3. Making an object type versionable




```
ProjectOffer::getTasks
entry//
// create a collection to be filled with connected tasks
create object instance connectedTasks of ReturnCollection;
// get all candidate version collections related to project offer
select many taskCVCs related by self->TaskCVC[R6.'tasks'];
for each cvc in taskCVCs
    // try to get a pinned version, if there is one
    select one pinnedTask related by cvc->Task[R4.'pinned version'];
    if not empty pinnedTask
        relate connectedTasks to pinnedTask across R8.'collection';
    else
        // there is no pinned version, get the latest version from the CVC
        select one latestVer related by cvc->Task[R5.'latest version']
        relate connectedTasks to latestVer across R8.'collection';
    end if;
end for;
return connectedTasks;
```

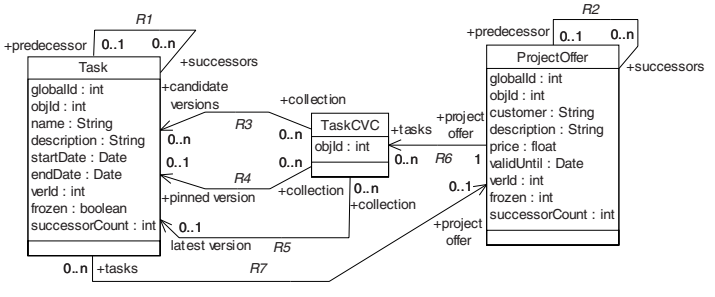
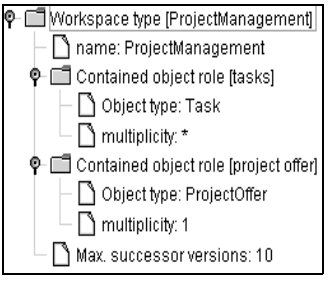


Fig. 4. Defining a relationship type

Balcer [12] for specifying the *createSuccessor* operation. Class definitions made in *Step A* are carried as data flows into *Step B*: This makes it impossible to define a non-existing object type as versionable. The specified constraint that limits the number of successor versions gets integrated into the procedure's actions.

3.3 Step C: Define Relationship Types

In *Step C*, we define relationship types. This includes the definition of *role names*, *multiplicities*, indication of *floating relationship ends* and *operation propagation* properties. In the example in Fig. 4, we define a relationship type *ProjectOffer-Task* with a floating relationship end for *tasks*. Creating a successor to a *project offer* version creates successors to related pinned *task* versions and freezing a *project offer* version freezes the pinned *task* versions (this behavior requires a refinement of *createSuccessor* and *freeze* operations that have been added to the model in *Step B*.) Partial results of the concrete transformation are illustrated in Fig. 4, which shows the class *TaskCVC* used for maintaining candidate version collections for *tasks* and the procedure *getTasks* that retrieves *tasks* connected to a *project offer*.




```
ProjectOffer::getTasks(wspGlobalId : int)
entry//
// create a collection to be filled with connected tasks
create object instance connectedTasks of ReturnCollection;
// get all candidate version collections related to project offer
select many taskCVCs related by self->TaskCVC[R6.'tasks'];
for each cvc in taskCVCs
    // find version related to the workspace
    select many candVersions related by cvc->Task[R3.'candidate versions']
    for each version in candVersions
        select any workspace related by version->ProjectManagement
            [R9.'project management'] where selected.globalId = wspGlobalId;
        if not empty workspace
            relate connectedTasks to version across R8.'collection';
            break;
        end if;
    end for;
end for;
return connectedTasks;
```

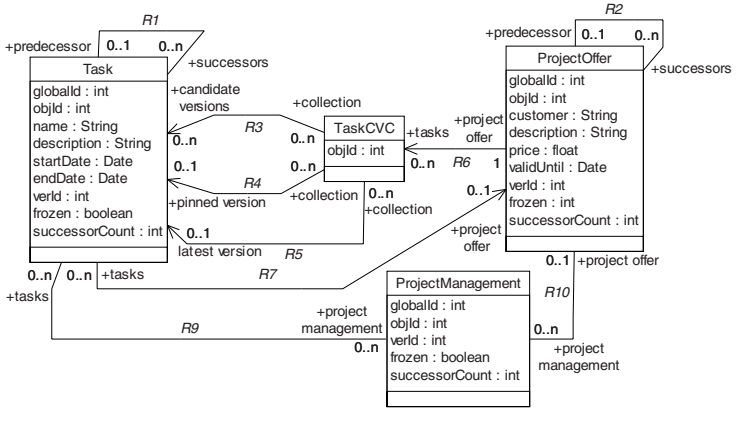


Fig. 5. Defining a workspace type

3.4 Step D: Define Workspace Types

In this step, the user defines workspace types. Each workspace type may be connected to regular object types using *attachment relationship types*. Traversal of a relationship is sensitive to the *globalId* of the workspace submitted to the navigation operation - only objects that are attached to this workspace can be reached across the relationship. Fig. 5 illustrates a definition of a workspace type *project management* that can attach one *project offer* and many *tasks* (but always only one version of each of them). The concrete transformation adds a class for the workspace type, the attachment relationship types and an operation for workspace-sensitive relationship traversal.

4 MT-Flow Templates and Workflow Models

Template-based approaches have become popular for dynamic generation of documents (like source code, tests, or documentation) that exhibit a common generic structure, but need to be instantiated many times with user-defined input values (a familiar example are *JavaServer Pages* that act as templates for HTML). Nearly all template-based approaches work in the same way. First, a template with integrated *control flow statements* that can repeat (or omit) some parts of the template is written. Second, a template engine replaces *placeholders* in a template with user-defined values (stored in a *context*) while executing the control flow. The template engine used in MT-Flow to generate concrete transformations expressed in XMI is *Velocity* [1]. An excerpt from the template used for *Step A* is illustrated in Fig. 6 (the '*$*'-signs represent references to values in the context).

MT-Flow's *workflow models* are acyclic directed graphs in which a node (called *configuration step*) represent (i) a specification of user-defined values, (ii) a production of concrete transformations from transformation templates, and (iii) an application of concrete transformations to the current model. The edges represent the flow of control and data among the steps. The *flow of control* defines the order in which configuration steps

```
...
<XMI.difference>
    <XMI.add href="currentModel.xmi|p1">
        <UML:Class xmi.id = '$Class.mId' name = '$Class.name' visibility = 'public'
            isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract = 'false' isActive = 'false'>
            <UML:Classifier.feature>
            #foreach( $item in $ClassAttributes )
                <UML:Attribute xmi.id = '$item.mId' name = '$item.name' visibility = 'private'
                    isSpecification = 'false' ownerScope = 'instance'>
                    <UML:StructuralFeature.type>
                        <UML:DataType xmi.idref = '$item.typeMId'/>
                    </UML:StructuralFeature.type>
                </UML:Attribute>
            #end
            </UML:Classifier.feature>
        </UML:Class>
    </XMI.add>
</XMI.difference>
...
```

Fig. 6. Transformation template for *Step A*

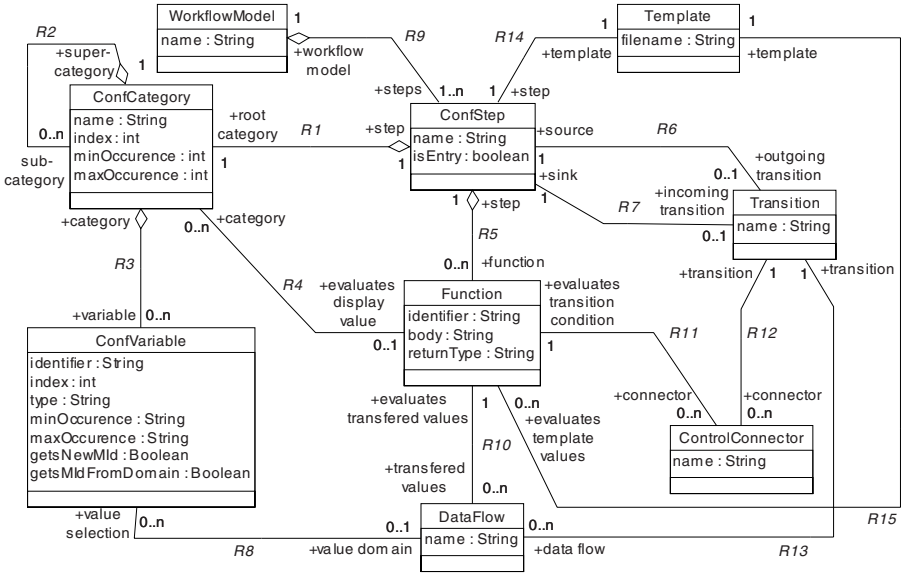


Fig. 7. Metamodel for MT-Flow's workflow models

are executed. The *flow of data* allows the exchange of configuration values among the steps. A *metamodel* for MT-Flow's workflow models is shown in Fig. 7.

A workflow model consist of many configuration steps, where one of the steps is marked as *entry*, denoting the beginning of execution. Each step defines a *root configuration category*, which may be subdivided into many *sub-categories*. A category may contain *configuration variables*, which take user values that will be used for producing a concrete transformation. These values are always of primitive types. Examples of categories and variables are illustrated throughout Figs. 2-5 (e.g., in Fig. 2, *Object type* is a category and *Name* is a variable). Categories and variables are indexed, which prescribes their ordering in the user interface. A configuration variable may be bound on one *data flow* received from the preceding configuration step (many data flows can be carried by a single transition). Such a data flow is then a set of values that are allowed to be selected for the variable (i.e., it defines the variable's domain). MT-Flow does not treat variables as scalars, but rather as *value wrappers*. A value wrapper consists of two components: a *value* (which is user-defined), and an *element identifier* associated with this value. Model elements need to provide unique identities so that succeeding model transformations can manipulate elements introduced by preceding model transformations. These identities can be created by MT-Flow automatically (using the setting *ConfVariable.getsNewMId*) or automatically adopted from a value wrapper in the value domain carried by a data flow (using the setting *ConfVariable.getsMIdFromDomain*).

There is a set of *functions* that can be associated with each step in a workflow model. These functions do not have side-effects and thus never change the state of value wrap-

pers. They are specified in MT-Flow's own language that supports easy scrolling through configuration categories and evaluation of arithmetic and string manipulation expressions. There is no global "*memory*" context for the workflow model, i.e., a function is allowed to access only (i) configuration variables, (ii) other functions defined in the same step and (iii) incoming data flows. There are four scenarios in which MT-flow's functions are used:

- *Evaluating a display value for a configuration category* (association *R4* in Fig. 7). A category can automatically display a value obtained as a function result, e.g., the category *Workspace type* in Fig. 5 automatically displays the value of its variable *name* in square brackets.
- *Evaluating value wrappers for a data flow* (association *R10* in Fig. 7). Each data flow is associated with a function that *returns a collection* of value wrappers to be carried by the data flow. These functions prepare values for domains used in the succeeding step.
- *Evaluating transition conditions* (association *R11* in Fig. 7). Each *control connector* is associated with exactly one function that returns a boolean value. An outgoing transition will be allowed only when all control connectors for the transition evaluate to *true* for the current configuration step. These functions imply the *allowed paths* in the workflow model that will be followed depending on the values of configuration variables. In this way, transition conditions implement *composition rules* (also known as *hard constraints*) for feature interdependencies [6], i.e., what features imply other features in the configuration. For example, in Fig. 4, only versionable object types can be selected for floating relationship ends in *Step C*. In case this constraint is violated, the transition to *Step D* will be prevented.
- *Evaluating values for a template* (association *R15* in Fig. 7). These functions abstract computation and string manipulation from the templates. The values they return can be accessed as variables in the template engine's *context* (see [1] for more information on the context) and are thus easily accessible from the templates.

5 Evaluation

We associate MT-Flow with a measure to estimate the benefits of automated model transformations. We *postulate* this measure as useful based on our experience with the product line for versioning systems, but have not observed its performance on a large set of various product lines yet.

The measure $XEPR_S(i)$ *(XMI Element Production Rate for a given system i)* is defined as

$$XEPR_S(i) = \frac{x(i)}{v(i)} \quad (1)$$

where $x(i)$ is the number of XMI elements present in the final model and $v(i)$ is the number of configuration value definitions and selections made by the user when configuring the system.

According to Czarnecki and Eisenecker [6], the structure of product-line members should not be treated as a configurability aspect of the product line. In other words, we should separate *concept instantiations* (e.g., definition of a new object type, relationship type, or workspace type) from *concept configurations* (e.g., definition of whether a relationship end is floating). If this was not the case, for example, one could define a versioning system with a large number of object types in *Step A* and easily arrive at a large $XEPR_S(i)$ value. Therefore, we define $XEPR_{SN}(i)$ as a normalization of the $XEPR_S(i)$ value using the count of concept instantiations $c(i)$ (for our product line for the versioning systems, this is the count of object types, attributes, relationship types, and workspace types defined for the system):

$$XEPR_{SN}(i) = \frac{XEPR_S(i)}{c(i)} \tag{2}$$

The *XEPR value for the product line* ($XEPR_{PL}$) is the average $XEPR_{SN}(i)$ value for n representative systems embraced in the product line.

$$XEPR_{PL} = \frac{\sum_{i=1}^{n} XEPR_{SN}(i)}{n} \tag{3}$$

The $XEPR_{PL}$ value for our example product line for versioning systems is 41,2.

Remark: $XEPR_{PL}$ is a *productivity measure*, since it compares the size of the final model with the user's effort for configuring the system. Its pitfall is that it assumes that in a manual modeling process, a manual insertion of every model element is associated with roughly the same effort. In our observation, this is not the case: Especially statements in action languages usually introduce a very large number of model elements (see examples in [15]) in proportion to the actual effort for defining a statement. A simple solution to this problem would be to assign weights to diverse model elements (based on practical experience with modeling efforts) and compute $x(i)$ in $XEPR_S(i)$ as a weighted sum.

6 Related Work

Peltier et al. [18] present *MTRANS*, which is an XSLT-based framework for model transformations. A special *rule-based language* is supported for describing the effect of transformations. Programs in this language get compiled into an XSLT transformation, which is used to map an input model expressed in XMI to an output model (also expressed in XMI). The idea is not described in the context of software product lines. Dependencies among sequentially applied transformations cannot be controlled.

There is a clear distinction among *specification refinement* approaches like *evolving algebras* (later renamed to *abstract state machines*) [9] or *especs* [17] and MT-Flow. Refinement approaches start with an abstract specification of a system which already includes key definitions of what the system does, but leaves out how it does this. For example, the initial specification may state that *sorting* is required, but the *definite sorting algorithm* is chosen in the succeeding refinements. *SPECWARE* [24] is an example

of a system supporting such *specification morphisms*. In MT-Flow, on the other hand, *new* structural and behavioral semantics can appear in the existing specification in each step. For this reason, the specification is not considered complete (even not on some abstract level) until the last configuration step has been performed. In this aspect, MT-Flow resembles approaches that support consecutive application of predefined patterns to automatically assemble the final application. An example of such an approach is *GREN*, described by Braga and Masiero [3]. The authors observe that when *instantiating software frameworks* to build concrete systems, the knowledge about flexible (variable) framework parts (also called the *hot spots*) is required. This problem is tightly related to software product lines, because most system families can be implemented using frameworks [6]. The authors propose a solution that relies on the sequential application of *patterns*, which is guided by a wizard and is allowed to follow only predefined paths. The allowed paths are described by associating different patterns in a formal model (the authors define a metamodel for such model). Concrete user-defined values for a selected pattern are not observed in a transition to the next pattern, as it is the case in MT-Flow. A direct relation to software product lines is not given, although the authors observe that in order for the approach to be successful, patterns have to be domain-specific.

7 Conclusion and Future Work

This paper presented MT-Flow, our approach for automatic construction of system specifications (expressed as models) within a software product line. The construction takes place by producing concrete model transformations from transformation templates and sequentially applying them to the current specification. The process is guided by MT-Flow's workflow model. In this way, we prove that defining a set of *domain-specific model transformation templates* and a *domain-specific workflow model* for modeling concrete systems from the product line represents a viable alternative to domain-specific languages for system configuration. We illustrated our approach on a product line for versioning systems. Based on our experience with this product line, we postulated a measure for estimating the benefits of MT-Flow for system configuration.

Our future work will be focused on the following areas.

- *Performance of the postulated measure*. We want to observe the performance of the measure postulated in Sect. 5 on a large set of software product lines. Our primary goal is to compare the obtained $XEPR_{PL}$ values to the effort of developing models manually (i.e. without MT-Flow).
- *Mining MT-Flow's transformation templates and workflow models*. Many companies sell a set of systems that actually belong to a software product line, without having a proper support for consistently specifying and afterwards generating individual systems on a basis of common architecture. We will try to prove that, with some human assistance, existing models of these systems (which have usually been developed from scratch) can be mined to obtain MT-Flow's transformation templates and the workflow model. In this way, MT-Flow can be used for developing further systems from the product line.

- *Relation to domain analysis*. At the moment, MT-Flow provides no direct relation to domain analysis approaches, like FODA (feature-oriented domain analysis) [10], which seem to be very important for analysis and understanding of variability within a product line. We will try to explore this relation and support it in the implementation of MT-Flow.

References

1. The Apache Jakarta Project: Velocity, http://jakarta.apache.org/velocity/
2. Bernstein, P.A.: Repositories and Object-Oriented Databases, SIGMOD Record 27:1 (1998), 34-46
3. Braga, R.T.V., Masiero, P.C.: Building a Wizard for Framework Instantiation Based on a Pattern Language, in: Proc. OOIS 2003, Geneva, Sept. 2003, 95-106.
4. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns, Addison-Wesley, 2002
5. Cummins Inc., http://www.cummins.com/
6. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications, Addison-Wesley, 2000
7. Frankel, D.: Model Driven Architecture: Applying MDA to Enterprise Computing, Wiley Publishing, 2003
8. Gamma, E., Helm, R., Johnson, R.E.: Design Patterns, Addison-Wesley, 1997
9. Gurevich, Y.: Evolving Algebra 1993: Lipari Guide, in: Specification and Validation Methods, Oxford University Press, 1995, 9-36
10. Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study, Tech. Report CMU/SEI-90-TR-21, Carnegie Mellon University, 1990
11. Kovse, J., Härder, T.: DSL-DIA - An Environment for Domain-Specific Languages for Database-Intensive Applications, in: Proc. OOIS'03, Geneva, Sept. 2003, 304-310
12. Mellor, S.J., Balcer, M.J.: Executable UML: A Foundation for Model-Driven Architecture, Addison-Wesley, 2002
13. MetaCase: MetaEdit+ Product Website, http://www.metacase.com/mep/
14. OMG: Model Driven Architecture (MDA) - A Technical Perspective, July 2001
15. OMG: Unified Modeling Language Specification, Version 1.5, Mar. 2003
16. OMG: XML Metadata Interchange Specification, Version 1.2, Jan. 2002
17. Pavlovic, D., Smith, D.R.: Composition and Refinement of Behavioral Specifications, Technical Report KES.U.01.6, Kestrel Institute, July 2001
18. Peltier, M., Bézivin, J., Guillaume, G.: MTRANS: A General Framework, Based on XSLT, for Model Transformations, Proc. workshop WTUMP 2001, Genova, Apr. 2001
19. Project Technology, Inc.: BridgePoint Development Suite, MC-2020 and MC-3020 Model Compilers, http://www.projtech.com/prods/index.html
20. Riehle, D., Fraleigh, S., Bucka-Lassen, D., Omorogbe, N.: The Architecture of a UML Virtual Machine, in: Proc. OOPSLA'01, Tampa, Oct. 2001, 327-341
21. Rumbaugh, J.E.: Controlling Propagation of Operations Using Attributes on Relations, in. Proc. OOPSLA'88, San Diego, Sept. 1988, 285-296
22. Saeki, M.: Toward Automated Method Engineering: Supporting Method Assembly in CAME, presentation at EMSISE'03 workshop, Geneva, Sept. 2003.

23. Simonyi, C.: The Death of Computer Languages, the Birth of Intentional Programming, Tech. Report MSR-TR-95-52, Microsoft Research, Sept. 1995
24. Srinivas, Y.V., Jullig, R.: Specware(tm): Formal Support for Composing Software, in: Proc. MPC'95, Irsee, July 1995.