Modeling Composition in Dynamic Programming Environments with Model Transformations

Uwe Zdun, Mark Strembeck

Institute of Information Systems, New Media Lab
Vienna University of Economics and BA, Austria
{uwe.zdun|mark.strembeck}@wu-wien.ac.at

Abstract Although dynamic programming environments are in widespread use, only basic runtime composition mechanisms are covered by today's modeling languages. Thus, it is common in real-world development projects that dynamic compositions are not modeled formally and are consequently hard to use, for example together with the model-driven paradigm where formal models are essential to generate source code. In this paper, we propose an approach based on model transformations between the valid structural and behavioral runtime states that a system can have. We use UML 2.0 class and activity diagrams for specifying the structural and behavioral model states and provide a UML 2.0 meta-model extension for describing the valid model transformations between corresponding model states.

1 Introduction

Each software composition mechanism defines the possible binding time(s) for the software elements it composes. The binding time is the point in time where the decision for a composition of particular software elements is made. Examples of binding times include development time, source instantiation time, source reuse time, build time, packaging time, installation time, start-up time, and execution time (runtime) [11].

Many different approaches exist to model software compositions that affect binding times before runtime. Examples for such approaches are UML class or component diagrams [21,23] and most architecture description languages (see, e.g., [12,2]). Some modeling languages also allow to specify runtime reconfigurations of components to a certain degree (see, e.g., [1,22]), but not beyond the level of changing the relationships of component or class instances. The specification of effects resulting from more sophisticated runtime composition mechanisms is only sparsely addressed in contemporary modeling languages.

At present, static programming languages such as Java, C++, or C# are still more prevalent than dynamic languages, such as CLOS, Perl, Python, Ruby, Smalltalk, or Tcl. However, together, the dynamic programming languages[1] have a substantial user base and are applied in a widespread application spectrum. Furthermore, some of the

[1] Not all dynamic composition mechanisms are directly realized as language features of programming languages – some are based on frameworks and tools. We thus use the more generic term *dynamic programming environments* below.

more static languages, like Java and C#, increasingly introduce dynamic language features such as limited forms of class reloading or reflection. In addition, aspect-oriented software composition frameworks [10] add language constructs that allow to produce similar effects to those of dynamic composition mechanisms. Furthermore, recent approaches also propose dynamic AOP features (see, e.g., [4,20]).

Given the broad use and increasing importance of dynamic composition mechanisms, it is obvious that modeling support for them is essential for the engineering, understanding, and maintenance of corresponding software systems. In case a software development project follows the model-driven paradigm [19] or the software factory approach [7], we even require a formal specification of the respective composition mechanisms. Such a formal definition is mandatory since it is impossible to generate source code from modeling level artifacts without a formal representation of model elements. Current modeling approaches, however, support dynamic software composition only to a minor degree. The UML 2.0, for example, does not support the specification of runtime changes of most UML modeling elements. For instance, in a class diagram it is not possible to model changing inheritance relations or the introduction of a new method to a class definition at runtime.

In this paper, we present an approach to model structural and behavioral system changes that result from the use of dynamic composition mechanisms. In particular, our approach is based on model transformations [3]. Because UML is the de-facto standard modeling language for software systems, we exemplify our approach by providing a UML meta-model extension (see [23]).

The remainder of this paper is structured as follows. In Section 2 we give a high-level introduction to our approach before we provide a detailed specification of Model Transformation Diagrams in Section 3. Subsequently, Section 4 presents an example for the use of Model Transformation Diagrams. In Section 5 we discuss related work before we conclude the paper and give an outlook on future work in Section 6.

2 Motivation and Approach Synopsis

In essence, this paper aims to provide a well-defined and widely applicable modeling approach to enable the systematic specification of dynamic changes in the structure of software systems as well as resulting changes in system behavior. From our experiences, it is equally important to model structural and behavioral changes, as they most often appear together and they represent two essential views to specify, comprehend, and maintain software systems.

Since the UML is by far the most important modeling language in the area of software engineering, we chose to define an extension to the UML 2.0 standard to realize our approach. However, the general approach does not depend on the UML and may also be realized with any other modeling language.

We especially aim to model a specific subset of the dynamic composition features that can be found in dynamic object-oriented programming environments: changes to structural object-oriented features of classes or components, and the behavior changes that result from them. Here the term "structural feature" relates to:

– the methods of a class,
– the fields of a class,
– the relationships of classes, such as superclass (generalization) relationships, dependencies (e.g. to an interface), associations, compositions, and aggregations,
– the relationships (e.g. the instance-of relationship) and slots of an instance, and
– the classes and objects defined in a system.

In addition, there are many other features that may be subject to dynamic composition, such as non-object-oriented structural features (e.g. procedures in procedural dynamic languages), data that is evaluated as code in homoiconic languages [9], or cross-cutting in aspect-oriented environments [10]. Even though these composition features might possibly be modeled with our approach, in this article we focus on the object-oriented features.

To further motivate our approach, let us consider a typical dynamic composition task that we also use as an example in Section 4. A storage interface abstracts a number of persistent storages, such as different databases. Objects can be made persistent using different persistence strategies that, in turn, must be configured with a storage to which they write the data. Any object can be made persistent or transient at any time. In a static programming environment we would need to instrument all classes that can potentially be made persistent. After that, we could turn persistence on and off at runtime. In a dynamic programming environment, however, we can perform the necessary changes at runtime. For instance, we can configure the storage dynamically with the persistence strategy, and then add the persistence class as a type to all objects (or classes) that should be made persistent. Unfortunately, these dynamic class changes cannot be modeled in most modeling languages.

Moreover, changing the class of an object usually has consequences for other structural elements. For example, fields that belong to the "old" class might get removed from instances, while other fields might be added. The behavior of the methods of the affected class changes as well. For instance, in the example above two different persistence strategies, e.g. eager persistence and lazy persistence, introduce new activities, and these activities are different for the two strategies. Again, switching between these behaviors cannot be modeled in most modeling languages. Furthermore, class changes might also have constraints. For instance, a persistence strategy must be associated with a storage (e.g. a flat file storage or a database), otherwise it must not be used for an object.

Similar concerns appear for all dynamic composition features listed above. Such features are, however, not well supported in contemporary modeling languages. Though, a static "snapshot" of the dynamic programming environment at a particular point in time can well be modeled using modeling languages like the UML. Our concept is thus to extend modeling languages so that legal snapshots of a system can be modeled to describe the valid states of a dynamic software system. To specify snapshot states of static system structures, we use UML class diagrams and variants of class diagrams, such as component diagrams[2]. UML activity diagrams are used to model system behavior and dynamic system facets.

[2] Please note that we allow the structure diagrams to contain instance specifications.

To describe changes in a system's structure or behavior we use model transformations. Our approach introduces a new type of UML diagram called *Model Transformation Diagram* (MTD). In essence, an MTD is a special type of state machine. Each MTD state includes a diagram that defines a valid structure or behavior specification of the system under consideration. The MTD also defines the possible changes of the system's structure or behavior as transformations of the model, and excludes changes that are not allowed. The details of MTD diagrams are defined in the following section.

3 Model Transformation Diagrams

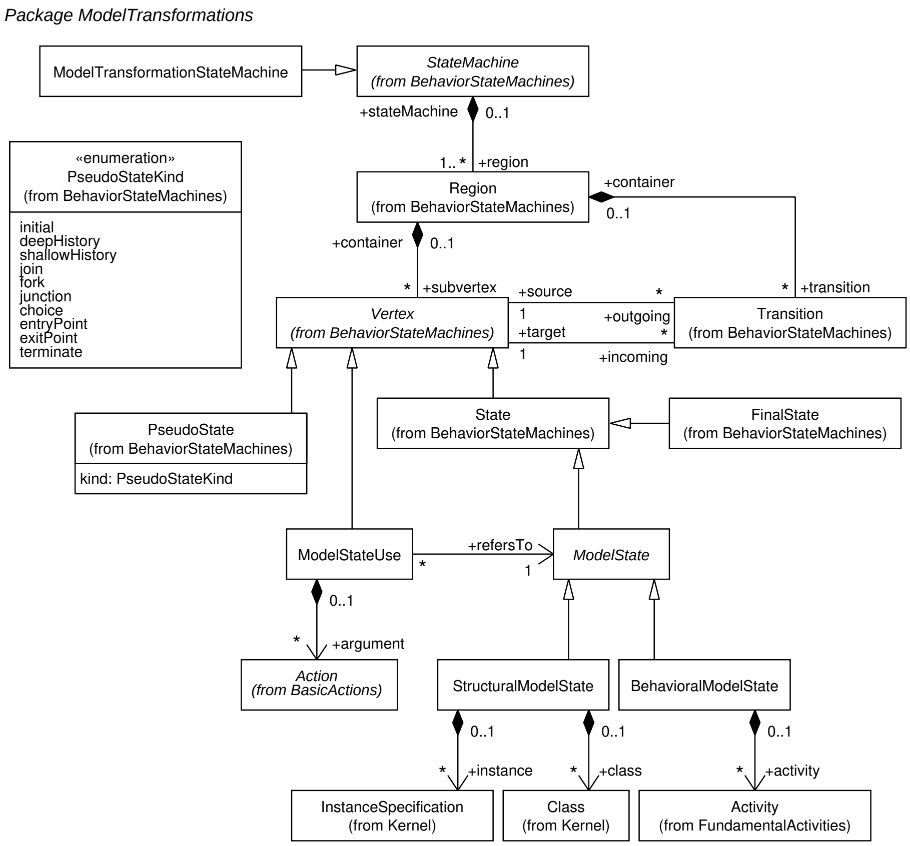


Figure 1. UML Meta-Model Extension for Model Transformation Diagrams

In this section, we describe our meta-model extension to the UML 2.0 standard. We introduce a new type of model called *Model Transformation Diagram* (MTD). To define MTDs formally, we specify the new package *ModelTransformations*. Figure 1 shows the meta-model for MTDs that constitutes the base model of the ModelTransformations package. Names of abstract classes are printed in italic letters, as customary in UML. Relevant UML2 classes from other packages are included in the figure (the "from" clause indicates the corresponding source package in the UML2 superstructure specification [23]).

NODE TYPE	NOTATION	Explanation & Reference
Model Transformation State Machine Frame	**mtd** name	Each Model Transformation State Machine is surrounded by a rectangular frame around the diagram. The compartment in the upper left corner contains the three letter token "mtd" and optionally the name of the state machine. See ModelTransformationStateMachine from ModelTransformations.
Structural Model State	**cd** name ClassName variable a variable b method x method y	Each Structural Model State is surrounded by a rectangular frame. The compartment in the upper left corner contains the token "cd" and optionally the name of the contained model. Each Structural Model State includes an UML2 class diagram or a variant of a class diagram, such as a component diagram. See StructuralModelState from Model-Transformations and Class from Kernel.
Behavioral Model State	**ad** name ActivityName	Each Behavioral Model State is surrounded by a rectangular frame. The compartment in the upper left corner contains the token "ad" and optionally the name of the contained model. Each Behavioral Model State includes an UML2 avtivity diagram or a variant of an activity diagram. See BehavioralModelState from Model-Transformations and Activity from FundamentalActivities.
Model State Use	**mref** name	A Model State Use refers to a Model State. The compartment in the upper left corner contains the token "mref". The rectangular frame contains the name of the model state it refers to. See ModelStateUse from ModelTrans-formations.

Figure 2. Basic notation elements for Model Transformation State Machines

The graphical notation of our model transformation diagrams is similar to UML2 interaction overview diagrams (cf. Figure 2), however, the MTD semantics differ significantly. The UML2 interaction overview diagrams are a variant of activity diagrams and describe the flow of control between different nodes, and each of these nodes is itself either an Interaction or an InteractionUse. In UML2 an Interaction is defined as a unit of behavior that focuses on the exchange of information between different model elements. Interactions are modeled using different types of diagrams, for example sequence diagrams or communication diagrams. An InteractionUse, on the other hand, refers to an Interaction. For details on interaction overview diagrams see [23].

In contrast to that, our Model Transformation Diagrams are a variant of state machines. Model transformation diagrams describe changes of the structural and behavioral specification of a software system. These changes are modeled through transitions between different diagrams. Therefore, model transformation diagrams may include two different types of states: each *structural model state* refers to an UML2 class diagram, and each MTD *behavior model state* refers to an UML2 activity diagram (see Figure 2).

As shown in Figure 1 the corresponding meta-model classes, StructuralModelState and BehavioralModelState, inherit from an abstract ModelState class, which is itself a State. A StructuralModelState aggregates elements of the types Class and InstanceSpecification, whereas a BehavioralModelState aggregates elements of the type Activity. Variants of the respective diagram types, such as component diagrams which specialize class diagrams, can thus also be contained in ModelStates.

In UML2, all elements of state machines that can have transitions are derived from the Vertex class. In addition to ordinary states, UML2 defines pseudo states (see the classes PseudoState and PseudoStateKind), such as initial, fork, join, choice, etc., as a subtype of Vertex. The UM2 FinalState class is a subtype of the State class. All Vertexes can be connected via Transitions (for additional details on state machines see [23]).

For the definition of MTDs we derive one more class from Vertex. This additional class is called ModelStateUse. Instances of ModelStateUse have no state themselves, so the class is directly derived from Vertex. A ModelStateUse refers to a ModelState, i.e. a ModelStateUse is purely a reference. It is used as a placeholder for the referred ModelState, which contains either a structural model state (modeled as a class diagram) or a behavioral model state (modeled as an activity diagram).

The ModelTransformationStateMachine is a state machine that contains MTDs. Like any other state machine it contains Vertexes and Transitions, which may be organized in Regions (see Figure 1). For the purposes of our ModelTransformations package, we need to constrain the ModelTransformationStateMachine so that it can only have vertexes of the types ModelState, ModelStateUse, FinalState, or PseudoState. That is, ordinary states must not be used in MTDs. The corresponding OCL constraint is given below:

```
context ModelTransformationStateMachine
inv: self.region->forAll(r | r.subvertex->forAll(v |
  v.oclIsKindOf(ModelState) or v.oclIsKindOf(ModelStateUse)
  or v.oclIsKindOf(FinalState) or v.oclIsKindOf(Pseudostate)))
```

The main transition type used in MTDs are *transform transitions*. Transform transitions express that the source model state of the transition is transformed to the target model state of the transition. Thus, transform transitions typically connect ModelStates and ModelStateUses. A transition from one model state to another means that the structure or behavior of a certain system aspect is transformed so that after the transition the system structure or behavior conforms to the state specified by the transition's target. A transform transition from an empty source model state to another target model state means that the model elements contained in the target are added to the system during the transformation.

To define transform transitions we extend the Transition class with the stereotype ≪*transform*≫ (see Figure 3). In principle, all transitions in MTDs are transform transitions. There are, however, some exceptions: most PseudoStates and FinalStates have no transform semantics, and are thus connected through ordinary transitions. For instance,

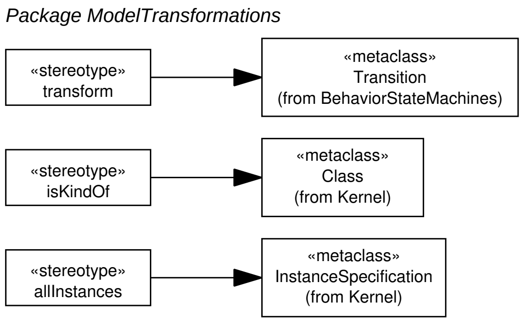


Figure 3. Stereotype Definitions for Model Transformation Diagrams

the "initial" PseudoState defines the starting point of a certain State Machine. Therefore, the transition from the "initial" PseudoState to a connected model state involves no transformations between model states. The following OCL constraint thus defines that all Transitions in a Model Transformation State Machine which are not connected to PseudoState or FinalState vertexes, must be typed with the ≪*transform*≫ stereotype:

```
context ModelTransformationStateMachine
inv: self.region->forAll(r | r.subvertex->forAll(v |
  v.incoming->forAll(t1:Transition|
    if (not v.oclIsKindOf(FinalState)) and
       (not v.oclIsKindOf(PseudoState)) then
     transform.baseTransition->exists(t2:Transition| t2 = t1))
  and
  v.outgoing->forAll(t1:Transition|
    if (not v.oclIsKindOf(PseudoState)) then
     transform.baseTransition->exists(t2:Transition| t2 = t1))
))
```

As mentioned above, PseudoStates cannot have transform transitions. There are, however, a few exceptions to this generic constraint. All exception cases are shown in Figure 4. The following OCL constraint defines that PseudoStates cannot be typed by the ≪*transform*≫ stereotype, except for the outgoing connections of "join", "fork", "junction", and "choice" PseudoStates:

```
context ModelTransformationStateMachine
inv: self.region->forAll(r | r.subvertex->forAll(v |
  if v.oclIsKindOf(PseudoState) then
    v.outgoing->forAll(t1:Transition|
      if not (v.kind = #join or v.kind = #fork or
              v.kind = #junction or v.kind = #choice)
      then not transform.baseTransition->exists(t2:Transition|
          t2 = t1))
    and
    v.incoming->forAll(t1:Transition|
      not transform.baseTransition->exists(t2:Transition|
          t2 = t1)))
```

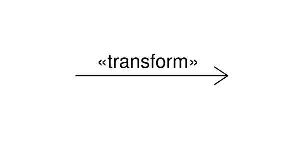
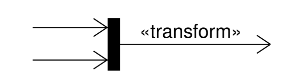
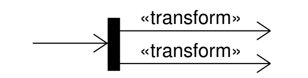
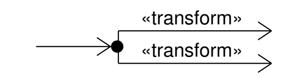
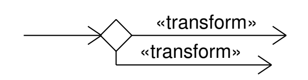
NODE TYPE	NOTATION	Explanation & Reference
Transform Transition		The transform transition is typed with the «transform» stereotype. Transform transitions connect ModelStates and ModelStateUses of the same kind. In some cases they can also be used with PseudoStates (see cases below). See Transition from BehaviorStateMachines and the stereotype «transform» from ModelTransformations.
Join Transform Transition		The outgoing transitions of "join" PseudoStates can be typed by the «transform» stereotype. See OCL constraints on ModelTransformation-StateMachine.
Fork Transform Transition		The outgoing transitions of "fork" PseudoStates can be typed by the «transform» stereotype. See OCL constraints on ModelTransformation-StateMachine.
Junction Transform Transition		The outgoing transitions of "junction" PseudoStates can be typed by the «transform» stereotype. See OCL constraints on ModelTransformation-StateMachine.
Choice Transform Transition		The outgoing transitions of "choice" PseudoStates can be typed by the «transform» stereotype. See OCL constraints on ModelTransformation-StateMachine.

Figure 4. Transform Transitions in Model Transformation Diagrams

Furthermore, to ensure that FinalStates never have incoming ≪*transform*≫ transitions we specify the OCL constraint shown below (remember that FinalState is *not* a PseudoState and has no outgoing transitions, see [23]):

```
context ModelTransformationStateMachine
inv: self.region->forAll(r | r.subvertex->forAll(v |
  if v.oclIsKindOf(FinalState) then
    v.incoming->forAll(t1:Transition|
      not transform.baseTransition->exists(t2:Transition|
          t2 = t1)))
```

All model states used within the same Model Transformation State Machine must be of the same kind because it is not sensible to describe a transformation from an activity diagram to a class diagram, or vice versa. Thus, each Model Transformation State Machine contains either structural model states or behavioral model states but not both. This is expressed by the following OCL constraints:

```
context ModelTransformationStateMachine
inv: self.region->forAll(r1 | r1.subvertex->forAll(v1 |
  v1.oclIsKindOf(StructuralModelState) or
    (v1.oclIsKindOf(ModelStateUse) and
     v1.refersTo.oclIsKindOf(StructuralModelState))
  implies
     self.region->forAll(r2 | r2.subvertex->forAll(v2 |
       (v2.oclIsKindOf(ModelState) implies
        v2.oclIsKindOf(StructuralModelState)) and
```

```
        (v2.oclIsKindOf(ModelStateUse) implies
         v2.refersTo.oclIsKindOf(StructuralModelState))))))

inv: self.region->forAll(r1 | r1.subvertex->forAll(v1 |
  v1.oclIsKindOf(BehavioralModelState) or
    (v1.oclIsKindOf(ModelStateUse) and
     v1.refersTo.oclIsKindOf(BehavioralModelState))
  implies
     self.region->forAll(r2 | r2.subvertex->forAll(v2 |
        (v2.oclIsKindOf(ModelState) implies
         v2.oclIsKindOf(BehavioralModelState)) and
        (v2.oclIsKindOf(ModelStateUse) implies
         v2.refersTo.oclIsKindOf(BehavioralModelState))))))
```

Finally, we define two more stereotypes for Class and InstanceSpecification from the Kernel package that can be used in structural model states (see also Figure 3):

– If a Class is typed by the ≪*isKindOf*≫ stereotype, it matches all classes that (directly or transitively) provide the type of the class that is labeled with the ≪*isKindOf*≫ stereotype.
– If an InstanceSpecification is typed by the ≪*allInstances*≫ stereotype, it matches all objects which are (direct or indirect) instances of the class that is labeled with the ≪*allInstances*≫ stereotype.

The ≪*isKindOf*≫ and ≪*allInstances*≫ stereotypes are mainly defined for convenience reasons, to allow for a compact specification of structural transitions (see also Section 4). These placeholder stereotypes especially ease situations where a StructuralModelState contains a class diagram that includes one or more class hierarchies. The use of these stereotypes, however, is optional to get smaller MTD models.

4 Example: Dynamic Composition of Persistence Strategies and Storages

In this section, we illustrate the use of MTDs via a number of dynamic composition functionalities of the scripting language XOTcl [17]. XOTcl is a dynamic language that supports dynamics in class relationships, superclass relationships, and mixin classes. Mixin classes [16] can be dynamically composed with any other class or object. A mixin class serves as a composition unit for a number of mixin methods. A mixin class is dynamically registered for an object or class as a *message interceptor*, which means that mixin classes intercept the method calls to the respective target object. Mixin classes are typically used as small building blocks to extend given classes [15]. These language functionalities are hard to model using standard UML diagrams because the corresponding dynamic changes in structure and behavior of the system cannot be captured.

As an example to demonstrate the use of MTDs, we consider the dynamic composition of persistence strategies for objects in XOTcl. The XOTcl library provides the class `Storage` as an abstract interface for a number of storage classes, and the class `Persistence` provides an abstract interface for two persistence strategies: eager and

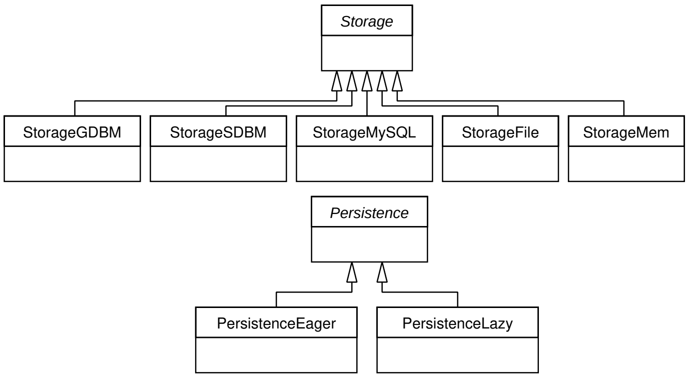


Figure 5. Classes implementing storages and persistence strategies

lazy persistence. The respective classes are shown in Figure 5. If an XOTcl object has a type relationship to one of the `Persistence` classes, the object is persisted to one of the storages defined by the `Storage` class. This can happen eagerly, i.e. all changes are immediately written to the storage, or lazily, i.e. the effect of all changes is written to the storage when the application closes down.

In our example, we now model the dynamic structural compositions that are valid for the composition of persistence strategies. First of all, we can make all instances of a particular class persistent. In XOTcl, two dynamic language elements can be used here: we can either add a `Persistence` class as superclass for the class whose instances should be made persistent, or we can add a `Persistence` class as a per-class mixin to the corresponding class. Figure 6 models these two situations in an MTD. The simple model state in the upper left corner shows a class diagram that matches all instances (indicated by the ≪*allInstances*≫ stereotype, see Section 3) of the type `Class` (note that `Class` is the type of all classes in the XOTcl object system). That is, the transformations can potentially be applied for all classes defined in XOTcl.

The other two model states depicted in Figure 6 show state transformations that can be applied to each XOTcl class. They show the possible combinations variants of `Persistence` classes and instances of `Class`. To include all subclasses of `Persistence` we add the stereotype ≪*isKindOf*≫ (see Section 3). This means any subclass of `Persistence` can be composed with all instances of `Class`, and XOTcl classes may either have a superclass relationship or a per-class mixin relationship to a respective `Persistence` class.

Similar to the example described above, individual objects can be dynamically composed with any `Persistence` subclass. The most general class in the XOTcl object system is the class `Object`. Thus, to make an instance of the `Object` class or an instance of a subclass of `Object` persistent, a `Persistence` class is either added as a class of the respective instance, or a `Persistence` class is added as a per-object mixin to the corresponding instance. Both transformations are depicted in Figure 7.

Finally, once a class or an object is made persistent, we must configure a persistent storage, so that the persistence strategy knows to which storage it can write the data

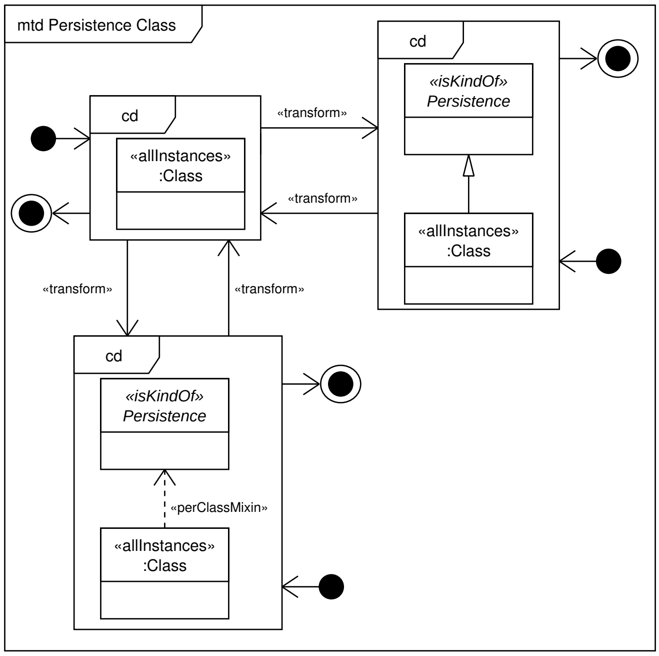


Figure 6. All possible compositions of the Persistence class and instances of Class

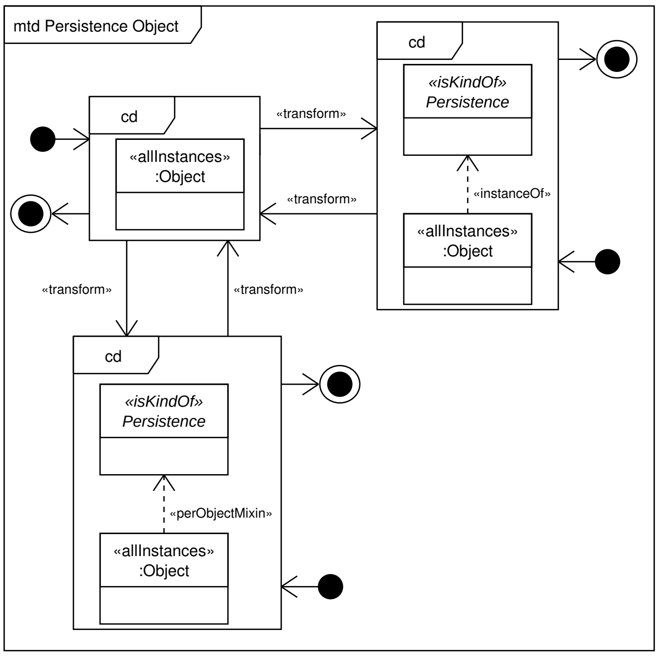


Figure 7. All possible compositions of the Persistence class and Object instances

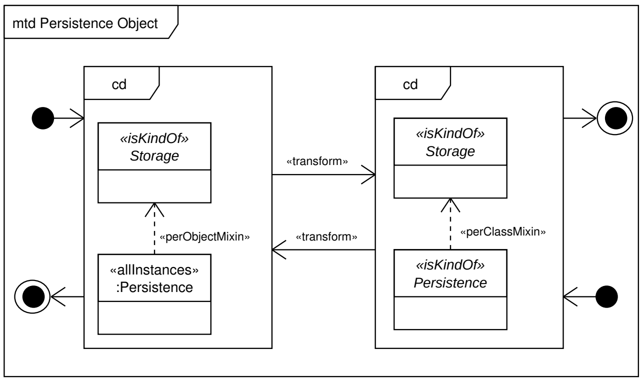


Figure 8. Possible compositions of the Persistence and Storage classes

(see also Figure 5). There are two mutual exclusive alternatives, but it is mandatory to select one of these alternatives. Figure 8 shows the two variants modeled via an MTD:

- The `Storage` is defined as per-class mixin for the `Persistence` class, meaning that all persistence data is written to the same storage.
- The `Storage` class is defined as per-object mixin for `Persistence` instances, meaning that the storage for each persistent instance is configured individually.

After we have defined the different structural transformations, we describe the corresponding model transformations of the behavioral model states. Figure 9 shows an empty activity diagram as an initial state. This initial state can either be transformed to a behavioral model state that introduces the eager persistence strategy or to a model state that introduces the lazy persistence strategy. The model states in Figure 9 are given as ModelStateUse references. The detailed behavioral model states for eager and lazy persistence that these ModelStateUse states refer to are shown in Figure 10.

5 Related work

The majority of existing architecture description languages (ADL) focus a static view on configurations [14]. Only a few ADLs, such as Rapide [12], support both static and dynamic views on the architecture, but do not support (dynamic) structure or behavior modification. The C2 ADL [13] can be seen as an exception because it allows for arbitrary modifications of the component and connector configuration. Similar to our approach, it uses a language for architecture modification (called AML). In contrast to MTDs, AML does not specify transformation paths, but a set of operations for insertion, removal, and rewiring of elements in an architecture at runtime.

Allen, Douence, and Garlan provide an extension to the architecture description language Wright [1]. This approach is closely related to our MTDs because it uses architectural snapshots to model static configurations, and special events triggering reconfiguration between these snapshots. The general idea to model dynamics is thus similar to the class diagram snapshots that we use in our MTDs.

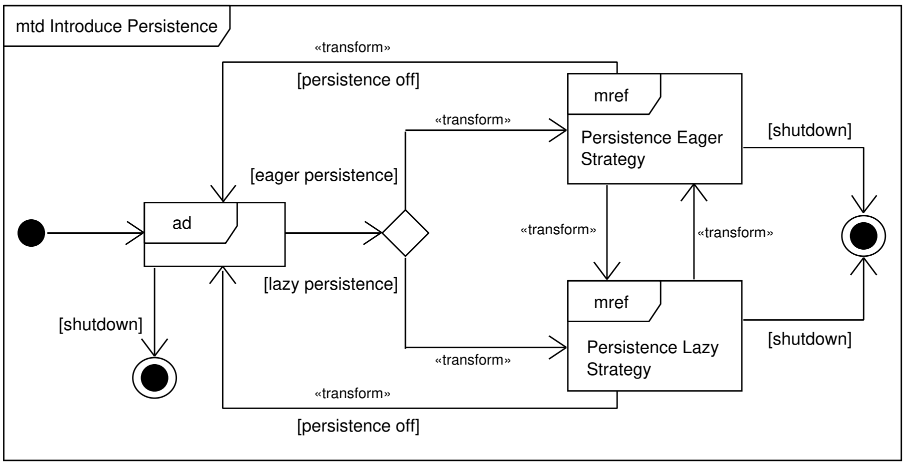


Figure 9. Behavioral model transformations for eager and lazy persistence

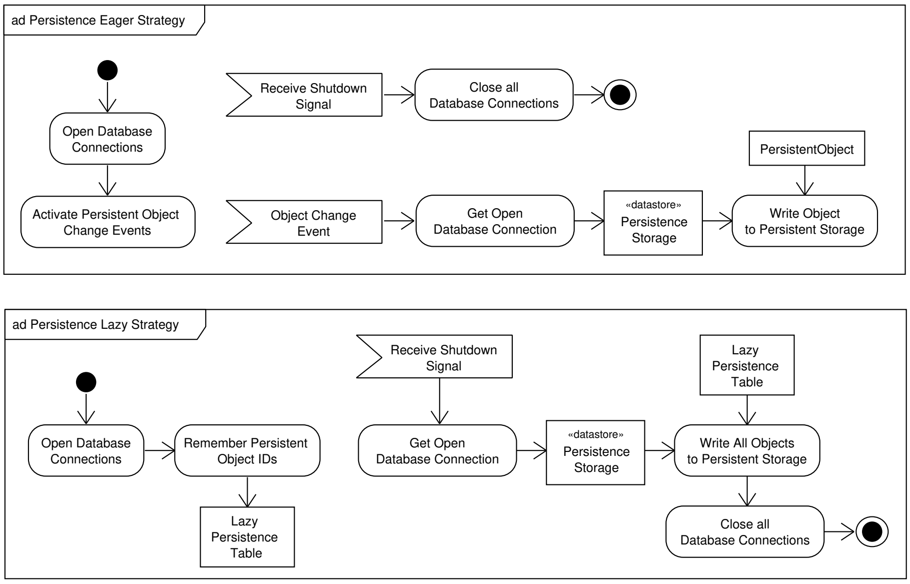


Figure 10. Detailed behavioral model states for eager and lazy persistence

In addition to the above mentioned approaches, there are a number of other approaches for modeling dynamics of software architectures. In [5] a recent survey of techniques for architectural reconfiguration is presented. While ADLs are often based on process algebras, other techniques used for specifying architectural reconfiguration are graph rewriting rules, graph transformation, and logic. None of the surveyed approaches, however, is based on model transformations like the approach presented in our paper.

A commonality of the approaches mentioned so far is that they focus on the addition and removal of components and connectors at runtime only. That means that, in contrast to our MTDs, those other approaches do not model other dynamic composition mechanisms. Moreover, corresponding changes in the behavioral model state which can be specified in MTDs via activity diagram snapshots, cannot be modeled using the above mentioned approaches.

Czarnecki and Antkiewicz propose an alternative way to model variants of behavioral models [6] that is comparable to our transformations of behavioral model states. The work described in [6] does, however, not cover the other elements of MTDs yet. In particular, Czarnecki and Antkiewicz use feature models to describe the possible variants of UML activity diagrams. Here a model is described via a model template, which specifies the possible composition of a system's features. Furthermore, they use a special-purpose tool to instantiate the model template from a feature configuration. Using the MTDs presented in this paper, we can use model transformations to add features in a similar fashion. In cases where many features need to be combined, the MTDs might get more complex than feature models. On the other hand, possible transformations of the models are not directly visible in feature models.

Our approach is based on the concept of model transformation. Recently, the research field of model-driven software development [19] has brought up a number of approaches for model transformations, mainly based on UML models (see, e.g., [3,18,24,7]). Our work extends these approaches with a concept for representing dynamic software compositions and with an extension of the UML standard for depicting structural and behavioral transformations suitable for these dynamic software compositions. As our general approach does not depend on a specific modeling language (as the UML for example), the transformation syntax and semantics in those other model transformation approaches could be extended, following our approach, to also support dynamic software composition. We have chosen the UML to exemplify our approach because it is the de-facto standard for software systems modeling.

Dynamic aspect-oriented approaches (see, e.g., [4,20,8]) provide an implementation of dynamic aspect-oriented transformations. However, modeling dynamic aspects is not yet in focus of the aspect-oriented community. Our approach can potentially be used to provide models for transformations implemented by dynamic aspect-oriented approaches. However, in this paper, we have only focused on modeling object-oriented language features.

6 Conclusion

In this paper, we have presented an approach to model structural and behavioral compositions in dynamic programming environments – with a special focus on object-oriented language features. Even though dynamic composition mechanisms are in widespread use, most contemporary modeling languages provide only little or even no support to specify dynamic compositions. Our paper describes an intuitive approach to resolve this problem. We use structural and behavioral snapshots of a system that are given as class and activity diagrams. These snapshots are interconnected using model transformations.

To be able to apply our approach in model-driven development, we introduced a formal meta-model extension to the UML. We chose the UML since it is a standardized modeling language that is in widespread use. Our general approach, however, is not depending on the UML. As a part of our future work, we plan to develop a model-driven tool-set for dynamic languages. So far our main focus was on structural evolution of dynamic object-oriented composition mechanisms. We plan to further extend our work in two directions: first, we want to introduce a pointcut language for model states to provide modeling support for (dynamic) AOP. Second, we will develop an approach to specify constraints on system states, e.g. via forbidden model states.

References

1. R. Allen, R. Douence, and D. Garlan. Specifying and analyzing dynamic software architectures. In *Proc. of the Conference on Fundamental Approaches to Software Engineering (FASE'98)*, Lisbon, Portugal, March 1998.
2. R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249, 1997.
3. J. Bezivin. From object composition to model transformation with the mda. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS USA)*, Santa Barbara, CA, USA, 2001. IEEE Press.
4. C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual Machine Support for Dynamic Join Points. In *AOSD 2004 Proceedings*. ACM Press, 2004.
5. J. S. Bradbury, J. R. Cordy, J. Dingel, and M. Wermelinger. A survey of self-management in dynamic software architecture specifications. In *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pages 28–33. ACM Press, 2004.
6. K. Czarnecki and M. Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In *Proc. of 4th International Conference on Generative Programming and Component Engineering (GPCE 2005)*, pages 422–437, Tallinn, Estonia, Sep/Oct 2005.
7. J. Greenfield and K. Short. *Software Factories: Assembling Applications with Patterns, Frameworks, Models & Tools*. J. Wiley and Sons Ltd., 2004.
8. R. Hirschfeld. AspectS – Aspect-Oriented Programming with Squeak. In *Objects, Components, Architectures, Services, and Applications for a Networked World*, LNCS 2591, pages 216–232. Springer-Verlag.
9. A. Kay. *The Reactive Engine*. PhD thesis, University of Utah, 1969.
10. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings European Conference on Object-Oriented Programming (ECOOP'97)*, pages 220–242, Finnland, June 1997. LCNS 1241, Springer-Verlag.

11. C. Krueger. Software product lines – binding times. http://www.softwareproductlines.com/introduction/binding.html, 2005.
12. D. C. Luckham and J. Vera. An event-based architecture definition language. *IEEE Trans. Softw. Eng.*, 21(9):717–734, 1995.
13. N. Medvidovic. Adls and dynamic architecture changes. In *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops*, pages 24–27. ACM Press, 1996.
14. N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, 26(1):70–93, 2000.
15. D. Moon. Object-oriented programming with flavors. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '86)*, volume 21 of *SIGPLAN Notices*, pages 1–8, Portland, November 1986.
16. G. Neumann and U. Zdun. Enhancing object-based system composition through per-object mixins. In *Proceedings of Asia-Pacific Software Engineering Conference (APSEC)*, Takamatsu, Japan, December 1999.
17. G. Neumann and U. Zdun. XOTcl, an object-oriented scripting language. In *Proceedings of Tcl2k: The 7th USENIX Tcl/Tk Conference*, Austin, Texas, USA, February 2000.
18. OMG. MOF 2.0 Query / Views / Transformations RFP. Technical Report ad/2002-04-10, Object Management Group, April 2002.
19. OMG. MDA Guide Version 1.0.1. Technical report, Object Management Group, 2003.
20. A. Popovici, T. Gross, and G. Alonso. Just In Time Aspects: Efficient Dynamic Weaving for Java. In *Proc. of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003)*, pages 100–109, Boston, USA, 2003. ACM Press.
21. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.
22. G. Succi, R. Wong, E. Liu, and M. Smith. Supporting dynamic composition of components. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, page 787, New York, NY, USA, 2000. ACM Press.
23. The Object Management Group. Unified Modeling Language: Superstructure. http://www.omg.org/technology/documents/formal/uml.htm, August 2005. Version 2.0, formal/05-07-04, Object Management Group.
24. D. Vojtisek and J.-M. Jzquel. MTL and Umlaut NG - Engine and framework for model transformation. *ERCIM News 58*, 58, 2004.