Reusable model transformation components with bentō

Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara

Modelling and Software Engineering Research Group (http://www.miso.es)
Universidad Autónoma de Madrid (Spain)

Abstract. Building high-quality transformations that can be used in real projects is complex and time-consuming. For this reason, the ability to reuse existing transformations in different, unforeseen scenarios is very valuable. However, there is scarce tool support for this task.
This paper presents bentō, a tool which supports the development and execution of reusable transformation components. In bentō, a reusable transformation is written as a regular ATL transformation, but it uses *concepts* as meta-models. Reuse is achieved by binding such concepts to meta-models, which induces the transformation adaptation. Moreover, composite components enable chaining transformations, and it is possible to convert an existing transformation into a reusable component. Bentō is implemented as an Eclipse plug-in, available as free software.

Keywords: Model transformation, Transformation reuse, Components, ATL

1 Introduction

Model transformation technology is the enabler of automation in Model-Driven Engineering (MDE), allowing model refactorings, optimizations, simulations and language conversions. However, developing a transformation from scratch is complex and error prone, even when specialized languages are used [6]. Thus, the reuse of existing high-quality transformations should be fostered, to amortize the effort invested in their development. One way to achieve this goal is to develop reusable transformation libraries, as it is common with general-purpose languages (e.g., ready to use Java libraries packaged as a .jar).

There are different reuse approaches for model transformations, ranging from reusing single rules (e.g., rule inheritance [12]) to reusing complete transformations (e.g., superimposition [11] or phasing [7]). However, most are type-centric, in the sense that a transformation cannot be reused for meta-models different from the ones used by the original transformation, thus limiting the reuse possibilities. There are some exceptions though, like [8] and [10], which use model subtyping and genericity respectively to define more reusable transformations. Other approaches [1] rely on transformation repositories and meta-model match and comparison techniques. However, they do not provide mechanisms to make transformations more reusable. Altogether, reuse of transformations is scarce in practice, as concluded in [3].

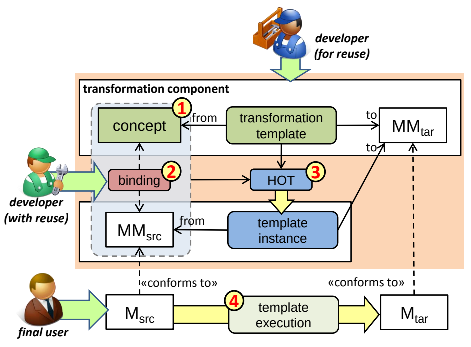


Fig. 1: Component instantiation

In the last few years, we have developed a transformation reuse approach inspired by generic programming [9] (e.g., templates in the C++ style) that we have implemented in a tool called `bentō`. The tool allows the definition of transformation components consisting of a transformation template, one or more concepts/meta-models, and a description of the component using a dedicated domain-specific language (DSL). *Concepts* are used as a means to describe the structural requirements that a meta-model needs to fulfil to allow the instantiation of the component with the meta-model. In particular, to instantiate the component for a meta-model, a binding mapping the concept elements to concrete meta-model elements (i.e., classes and features) should be written using another DSL. This binding adapts the transformation template to yield a new transformation ready to use with the concrete meta-model. Fig. 1 shows this process. In addition, composite components permit combining simpler components using transformation chaining.

Our approach has advantages w.r.t. existing proposals: (i) it is more flexible, since it permits applying components for meta-models that are structurally very dissimilar to the concept; (ii) it does not require adapting the bound meta-models and their instance models, but our template rewriting approach generates a new transformation that can be readily applied to them, improving performance; (iii) no special traceability handling is needed; and (iv) our component model allows the precise description of components and provides a systematic way of reuse.

The aim of this tool-demo paper is to describe the architecture of `bentō` and its features from the perspective of the tool user. A summary of the concrete demo presented at the conference is available online[1]. The concepts behind the component model underlying `bentō` have been reported elsewhere [4, 5]. Nevertheless, the tool has been improved since its first versions with new features such as support for in-place transformations, validation, integration with a static analyser [6][2], and a REST-based repository to store and retrieve components.

Paper Organization. Sec. 2 overviews `bentō`'s architecture, and the following ones show its main use cases: developing reusable components (Sec. 3), reusing components (Sec. 4), making a reusable component out of an existing transformation (Sec. 5), and selecting components (Sec. 6). Sec. 7 finishes with the conclusions and future work.

[1] Summary of the demo: `http://www.miso.es/tools/bento_demo_icmt2015.pdf`
[2] anATLyzer: `http://www.miso.es/tools/anATLyzer.html`

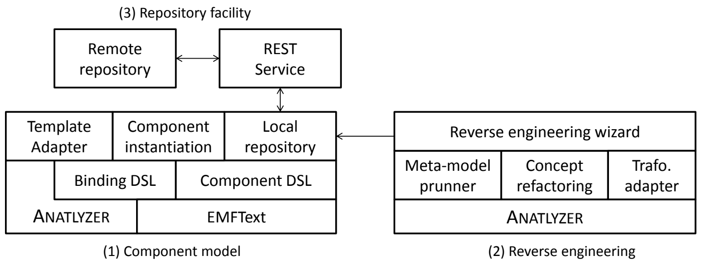


Fig. 2: bentō architecture

Dimension	**bentō feature**	**Description**
Abstraction	Concept	Plain Ecore meta-model with optional annotations
Specialization	Binding Template adaptation Binding validator	A DSL to map concepts and meta-models HOT to rewrite a template according to a binding It validates the syntactic correctness of bindings
Selection	Tags, documentation Repository Existing artefacts	Markdown documentation and attached tags REST-based repository and search wizard Reverse engineering process supported by a wizard
Integration	Component definition Standard structure Composite components	A DSL to define components and their dependencies Structure and local installation of components Aggregated components

Table 1: Features of bentō

2 Tool architecture

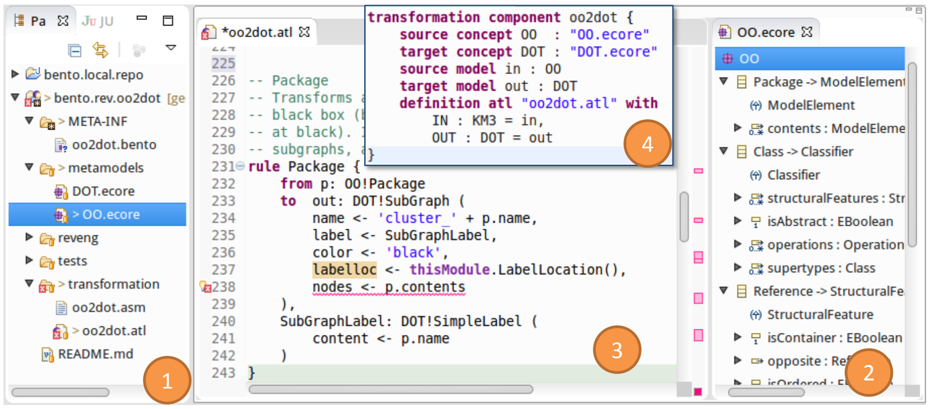
Bentō is an Eclipse-plugin. Its architecture, depicted in Fig. 2, consists of a component model, a reverse engineering wizard, and a remote repository facility. Implementation-wise, the two main elements of the component model are anATLyzer to statically analyse ATL transformations, and the *Template Adapter* which is able to solve non-trivial heterogeneities between concepts and meta-models (see Secs. 3 and 4). The DSLs to specify components and bindings has been defined using EMFText. In addition, bentō includes a reverse engineering wizard to convert an existing transformation into a reusable component (see Sec. 5), and a REST-based repository to share components (see Sec. 6).

As stated by Krueger [2], the practical use of components should consider four dimensions: abstraction, specialization, selection and integration. Table 1 summarizes the features of bentō according to these dimensions.

3 Developing components

As a running example, let us consider the visualization of object-oriented models by means of a transformation to the DOT format. This transformation will be similar for a range of object-oriented languages such as Ecore, KM3, UML or even Java. Hence, we create a reusable transformation component called oo2dot that can be specialized for such languages.

A transformation component is made of a transformation template, one or more concepts or meta-models over which the template is defined, and a descrip-



```
bento.local.repo
> bento.rev.oo2dot [ge
  > META-INF
    oo2dot.bento
  > metamodels
    DOT.ecore
    > OO.ecore
  reveng
  tests
  > transformation
    oo2dot.asm
    > oo2dot.atl
  README.md
```

```
-- Package
-- Transforms
-- black box (
-- at black).
-- subgraphs,
rule Package {
    from p: OO!Package
    to  out: DOT!SubGraph (
        name <- 'cluster_' + p.name,
        label <- SubGraphLabel,
        color <- 'black',
        labelloc <- thisModule.LabelLocation(),
        nodes <- p.contents
    ),
    SubGraphLabel: DOT!SimpleLabel (
        content <- p.name
    )
}
```

```
transformation component oo2dot {
    source concept OO  : "OO.ecore"
    target concept DOT : "DOT.ecore"
    source model in : OO
    target model out : DOT
    definition atl "oo2dot.atl" with
        IN : KM3 = in,
        OUT : DOT = out
}
```

```
OO
  Package -> ModelElement
    ModelElement
    contents : ModelEleme
  Class -> Classifier
    Classifier
    structuralFeatures : Str
    isAbstract : EBoolean
    operations : Operation
    supertypes : Class
  Reference -> StructuralFe
    StructuralFeature
    isContainer : EBoolean
    opposite : Re
```

Fig. 3: Definition of component in bentō

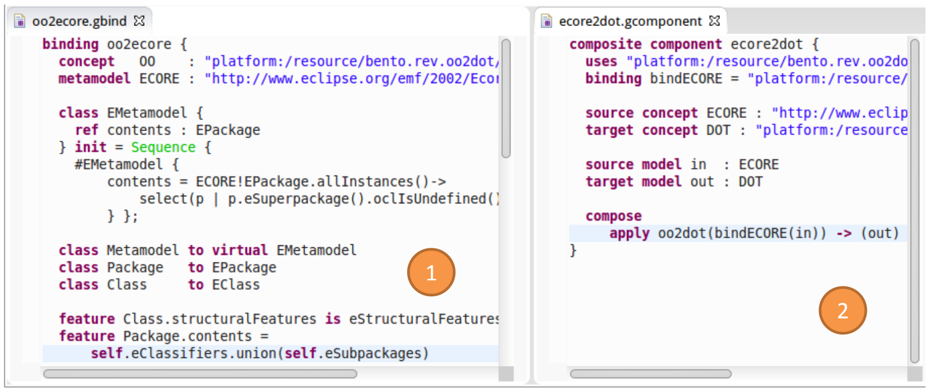
tion of the component. This is shown in Fig. 3. These artefacts are organized according to the structure shown in (1). In this case, the transformation has a source concept (OO.ecore) and a target meta-model (DOT.ecore). A concept is just a regular Ecore meta-model (2), but it only contains the elements required by the transformation, thus removing "accidental elements" for this particular scenario like configuration attributes (e.g., transient in Ecore) or features that we do not intend to visualize (e.g., annotations in Ecore). The transformation template is a regular ATL transformation. Moreover, bentō uses anATLyzer to statically analyse the transformation templates in order to provide some guarantee of their correctness, as illustrated by the error markers in (3). The component specification, shown in (4), describes the inputs and outputs of the transformation, since it is a single component.

Components can be exported to a remote component repository using the Eclipse export menu (see more details in Sec. 6).

4 Reusing components

In order to instantiate a component for a concrete meta-model, the component must be specialized by defining a binding from the elements in the concept to elements of the meta-model. Fig. 4(1) shows part of the binding from the OO concept to the Ecore meta-model. The binding is used to automatically rewrite the original template, so that it becomes able to transform models conforming to the bound meta-model. A distinguishing feature of our tool is that it allows sophisticated adaptations that bridge many heterogeneities between the concept and the meta-model. This is possible due to the precise typing information gathered by anATLyzer. A detailed account of the binding features and solvable heterogeneities is given in [4].

Fig. 4(2) shows how to instantiate and execute a component. We need to define a composite component which imports the component to instantiate (oo2dot)



```
binding oo2ecore {
  concept   OO    : "platform:/resource/bento.rev.oo2dot/
  metamodel ECORE : "http://www.eclipse.org/emf/2002/Ecor

  class EMetamodel {
    ref contents : EPackage
  } init = Sequence {
    #EMetamodel {
        contents = ECORE!EPackage.allInstances()->
            select(p | p.eSuperpackage().oclIsUndefined()
        } };

  class Metamodel to virtual EMetamodel
  class Package   to EPackage
  class Class     to EClass

  feature Class.structuralFeatures is eStructuralFeatures
  feature Package.contents =
      self.eClassifiers.union(self.eSubpackages)
```

```
composite component ecore2dot {
  uses "platform:/resource/bento.rev.oo2do
  binding bindECORE = "platform:/resource/

  source concept ECORE : "http://www.eclip
  target concept DOT : "platform:/resource

  source model in  : ECORE
  target model out : DOT

  compose
     apply oo2dot(bindECORE(in)) -> (out)
}
```

Fig. 4: Binding and composite component definition in bentō

and the binding, and uses the apply command to adapt the component according to the binding and execute it on the given source/target models. Composite components also support sequencing components to create transformation chains.

5 Reverse engineering existing transformations

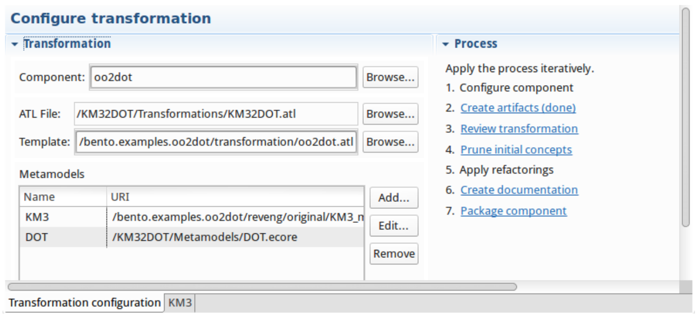
To enable the reuse of existing ATL transformations, bentō provides a reverse engineering facility that converts a transformation whose meta-models are "hard-coded" into a concept-based transformation component. This facility uses anATLyzer to statically determine the elements of the original meta-models that the transformation does not use, and then, it extracts a concept where such elements are pruned. In the process, a set of automated or manual refactorings can be applied to improve the quality of the extracted concept, which may imply the automatic co-evolution of the transformation.

From the user perspective, there is a wizard to configure the process, apply refactorings and automatically generate the component specification.

In the running example, instead of developing the oo2dot transformation from scratch, we could convert the KM32DOT transformation available in the ATL transformation zoo into a reusable component. This transformation has 418 LOC, 18 helpers and 7 rules; thus, its reuse saves a lot of effort. Fig. 5 shows the wizard to configure the conversion, which includes links to guide the steps to perform.

6 Selecting components

The ability to search and select components is important in any reuse approach, being typically enhanced by concise abstractions that can be easily understood and compared [2]. In our case, given a transformation component, it is easy



Configure transformation

Transformation

Component: oo2dot

ATL File: /KM32DOT/Transformations/KM32DOT.atl

Template: /bento.examples.oo2dot/transformation/oo2dot.atl

Metamodels

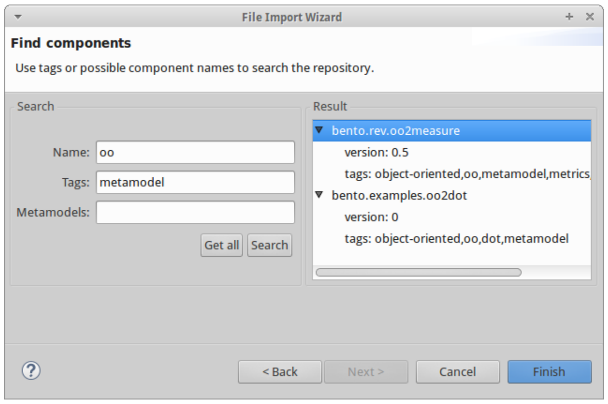
Name	URI
KM3	/bento.examples.oo2dot/reveng/original/KM3_n
DOT	/KM32DOT/Metamodels/DOT.ecore

Process

Apply the process iteratively.

1. Configure component
2. Create artifacts (done)
3. Review transformation
4. Prune initial concepts
5. Apply refactorings
6. Create documentation
7. Package component

Fig. 5: Reverse engineering of KM32DOT



Find components

Use tags or possible component names to search the repository.

Search

Name: oo

Tags: metamodel

Metamodels:

Result

- bento.rev.oo2measure
  - version: 0.5
  - tags: object-oriented,oo,metamodel,metrics
- bento.examples.oo2dot
  - version: 0
  - tags: object-oriented,oo,dot,metamodel

Fig. 6: Searching the repository by name and tags

to examine its concepts (i.e., its interface) to decide whether they match the meta-models at hand.

In addition, to facilitate the publication and retrieval of components, we have implemented a simple REST service to publish and search components. Components may have tags attached, which can be used in the search. Once a component is selected, it is automatically installed in a local project (bento.local.repo) and can be referenced by other projects using the URI bento:/componentName. When a component uses a URI of this kind, if the corresponding component has not already been installed, it is automatically sought in the remote repository by name. This feature is akin to *Maven* dependency resolution, and is intended to facilitate the maintenance of composite components. Fig. 6 shows the Eclipse import wizard to search and install components.

7 Conclusions

In this paper, we have presented bentō, a tool supporting model transformation components. It includes features like flexible template adaptations, reverse engineering of existing transformations into reusable components, a REST-based repository and component validations. To the best of our knowledge, this is the first component model for model transformations.

Bentō is available as free software (`http://github.com/jesusc/bento`) and as a ready to install Eclipse-plugin (`http://www.miso.es/tools/bento.html`).

Currently, Java programs can be packaged as `bentō` components, but these cannot be adapted. We are working on the possibility to package and adapt other MDE artefacts as `bentō` components, like Acceleo generators.

Acknowledgements. This work was supported by the Spanish Ministry of Economy and Competitivity with project Go-Lite (TIN2011-24139), the R&D programme of the Madrid Region with project (SICOMORO S2013/ICE-3006), and the EU commission with project MONDO (FP7-ICT-2013-10, #611125).

References

1. F. Basciani, D. D. Ruscio, L. Iovino, and A. Pierantonio. Automated chaining of model transformations with incompatible metamodels. In *MODELS*, volume 8767 of *LNCS*, pages 602–618. Springer, 2014.
2. C. W. Krueger. Software reuse. *ACM Comput. Surv.*, 24:131–183, 1992.
3. A. Kusel, J. Schönböck, M. Wimmer, G. Kappel, W. Retschitzegger, and W. Schwinger. Reuse in model-to-model transformation languages: are we there yet? *SoSyM*, 2013.
4. J. Sánchez Cuadrado, E. Guerra, and J. de Lara. A component model for model transformations. *IEEE Transactions on Software Engineering*, 40(11):1042–1060, 2014.
5. J. Sánchez Cuadrado, E. Guerra, and J. de Lara. Reverse engineering of model transformations for reusability. In *ICMT 2014*, volume 8568 of *LNCS*, pages 186–201. Springer, 2014.
6. J. Sánchez Cuadrado, E. Guerra, and J. de Lara. Uncovering errors in ATL model transformations using static analysis and constraint solving. In *25th IEEE ISSRE*, pages 34–44, 2014.
7. J. Sánchez Cuadrado and J. G. Molina. Modularization of model transformations through a phasing mechanism. *SoSyM*, 8(3):325–345, 2009.
8. S. Sen, N. Moha, V. Mahé, O. Barais, B. Baudry, and J.-M. Jézéquel. Reusable model transformations. *SoSyM*, 11(1):111–125, 2010.
9. A. Stepanov and P. McJones. *Elements of Programming*. Addison Wesley, 2009.
10. D. Varró and A. Pataricza. Generic and meta-transformations for model transformation engineering. In *UML'04*, volume 3273 of *LNCS*, pages 290–304. Springer, 2004.
11. D. Wagelaar, R. V. D. Straeten, and D. Deridder. Module superimposition: a composition technique for rule-based model transformation languages. *SoSyM*, 9(3):285–309, 2010.
12. M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, W. Schwinger, D. S. Kolovos, R. F. Paige, M. Lauder, A. Schürr, and D. Wagelaar. Surveying rule inheritance in model-to-model transformation languages. *JOT*, 11(2):3: 1–46, 2012.