

# Language Workbench Contest 2011

## EMFText, JaMoPP Solution

Florian Heidenreich, Jendrik Johannes, Sven Karol,  
Mirko Seifert, and Christian Wende

Institut für Software- und Multimediatechnik  
Technische Universität Dresden  
D-01062, Dresden, Germany  
{florian.heidenreich, jendrik.johannes, sven.karol,  
mirko.seifert, c.wende}@tu-dresden.de

**Abstract.** This document contains a description of a solution to the Language Workbench Competition 2011 built using the tools EMFText and JaMoPP. It may serve as documentation for the EMFText tool and can be used in particular by users that are new to EMFText or who want to compare this solution with the ones provided by other language workbenches.

## 1 Preliminaries

To explore this solution in detail, you need to install the latest version of EMFText from the EMFText update site<sup>1</sup>. This document was created at the time EMFText 1.3.1 was the latest release. Later releases may expose different behavior. We'll try to keep this document up to date.

The source code of all the languages described in the following are available from the EMFText Subversion repository<sup>2</sup>. The languages start with the prefix `org.emftext.language.lwc11` and can be found in folder `trunk/EMFText/Languages/`.

## 2 Phase 0 - Basics

According to the task description of the Language Workbench Competition, this section shows basic features of EMFText. This includes building a simple structural DSL to define entities (Sect. 2.1), defining constraints on this language (Sect. 2.3) and splitting models into multiple files while cross-referencing elements in other files (Sect. 2.4). Generating code from DSL models is not supported by EMFText. Nonetheless, we'll show how to use other EMF-based tools that are especially suited for code generation in conjunction with the entity DSL (Sect. 2.2).

---

<sup>1</sup> <http://www.emftext.org/update>

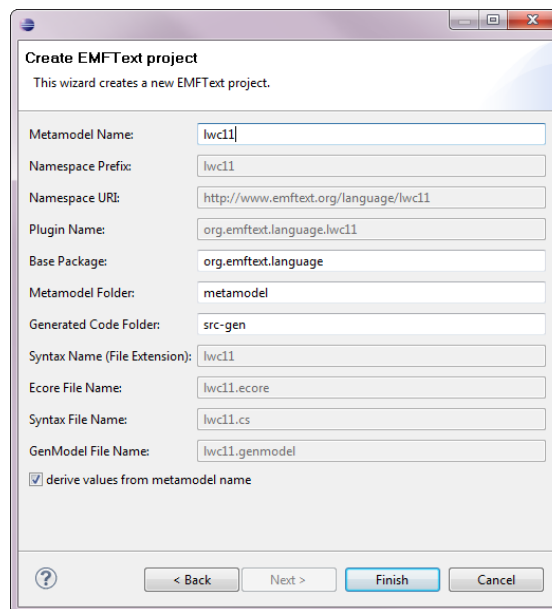
<sup>2</sup> <http://svn-st.inf.tu-dresden.de/svn/reuseware>

## 2.1 Phase 0.1 Simple (structural) DSL

To create new DSLs from the scratch, EMFText provides a wizard, which can be invoked by selecting **File -> New -> EMFText project**. The wizard allows to specify the following parameters to start the DSL development:

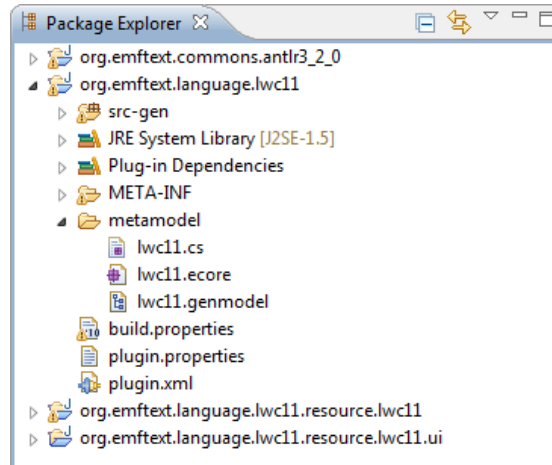
- the name of the new metamodel
- the namespace prefix for the new metamodel
- the namespace URI for the new metamodel
- the name of the plug-in that will contain the metamodel
- the base package for the generated metamodel code
- the folder the metamodel (i.e., the `.ecore` and `.genmodel` file) and the syntax (i.e., the `.cs` file) will be stored in
- the name of the source folder to store the generated metamodel code in
- the file extension which shall be used for the files containing the textual DSL models
- the names of the Ecore file, the syntax definition file and the generator model

Usually it is sufficient to specify only a subset of these parameters and let EMFText derive the others. If you need to make more custom settings, just disable the `derive values from metamodel name` checkbox. In Fig. 1 the wizard completed with the settings we have used for the Entity DSL is shown.



**Fig. 1.** Creating a new project using the wizard

After pushing the **Finish** button, EMFText will create four plug-ins yielding the workspace shown in Fig. 2. The workspace does contain one plug-in that provides the parser runtime (`org.emftext.commons antlr3_2_0`), one plug-in that hosts the metamodel (`org.emftext.language.lwc11`) and two plug-ins the implement the DSL functionality (`...lwc11.resource.lwc11` and `...lwc11.resource.lwc11.ui`).



**Fig. 2.** Workspace after creating the new project

EMFText does not require an existing metamodel before one can start defining the textual syntax for the language. Therefore, we need to create an `.ecore` model for the DSL. We used the sample tree editor provided by EMF, but other editors (e.g., the TextEcore DSL editor<sup>3</sup>) can be used as well.

We decided to split the metamodel into two files (i.e., `lwc11.ecore` and `types.ecore`) to allow the creation of type libraries independent of the entity models. The definition of these two metamodels is straightforward<sup>4</sup>. To model types, we define an abstract metaclass `NamedElement`, a metaclass `Type`, which subclasses `NamedElement` and a further subclass `PrimitiveType`. Also, a metaclass `TypeLib` models a container for a set of types. To model entities, we use the metaclass `Entity`, which inherits from `Type` as entities are supposed to function as types. Each `Entity` holds a set of features, whereas an `EntityModel` can be used to group multiple `Entity` elements. The resulting metamodels are shown in Fig. 3.

After designing the metamodel for types and entities, the actual syntax development with EMFText can start. The new project wizard did already create a sample `.cs` file, which needs to be adjusted to our new metamodels. To define

<sup>3</sup> <http://www.emftext.org/language/textecore>

<sup>4</sup> The types created by the new project wizard need to be removed from `lwc11.ecore`

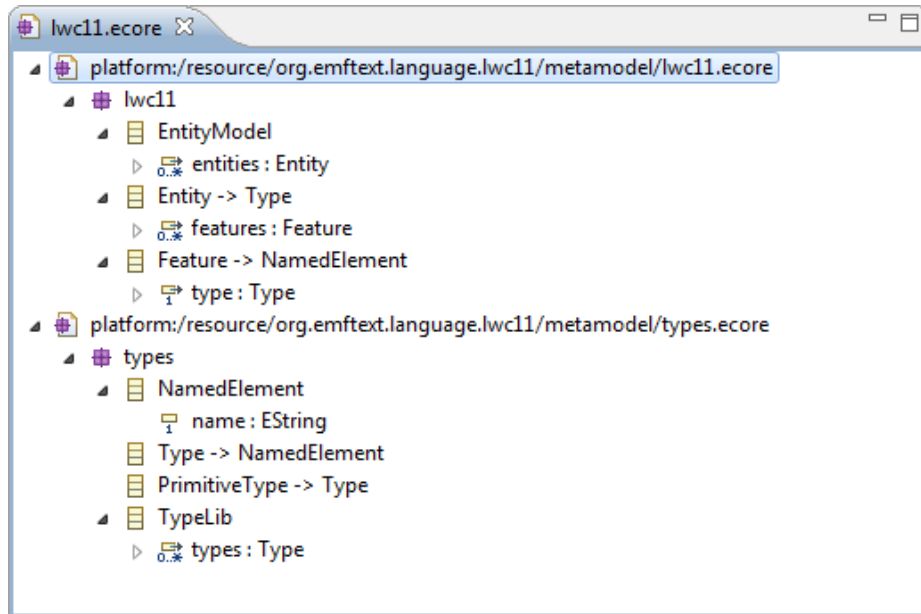


Fig. 3. Initial metamodel of the Entity DSL

syntax for entities, three basic syntax rules are sufficient. These are shown in Listing 1.1.

Listing 1.1. Initial syntax for the Entity DSL

```
SYNTAXDEF lwc11
FOR <http://www.emftext.org/language/lwc11>
START EntityModel

RULES {
  EntityModel ::= ("import" imports['<', '>'])* entities*;
  Entity      ::= "entity" name [] "{" features* "}";
  Feature     ::= type [] name [];
}
```

An EMFText syntax specification starts with a declaration of the file extension which will be used for the DSL models (line 1). This is followed by a reference to the metamodel we'd like to define syntax for (line 2). We do not need to explicitly tell EMFText where to find this metamodel, because the name of the `.genmodel` file is equal to the name of the syntax specification. Note that EMFText references the generator model instead of the `.ecore` model, because it requires information about the generated metamodel code, which is only available in the former model. In line 3 the start symbol is set to be `EntityModel`. Objects of this class will form the root of the Entity DSL models.

Then, syntax is defined for the three metaclasses `EntityModel`, `Entity` and `Feature` (lines 6 to 8). The syntax for objects of type `EntityModel` starts with an optional list of imports—each represented by the keyword `import` and a URI enclosed in angle brackets. These are not required yet, but we'll need them in Sect. 2.4.

Then, text for the contained entities (`entities*`) can follow. Entities themselves are represented by the keyword `entity`, followed by their name and the set of contained features enclosed in curly brackets. The syntax for the name attribute is defined as `name[]`, which states that names must adhere to the predefined token `TEXT`, which is similar to Java identifiers (i.e., it must not contain spaces). Finally, the syntax for features is defined to consist of the type and the name of the feature. Again, the `TEXT` token is implicitly used here. As `type` is a non-containment reference, we can see that both `EAttributes` (e.g., `name`) and non-containment `EReferences` must specify the token which can be used to represent them. Containment references do not require this, since they are represented by a syntax rule (i.e., the rule for the type of the reference) rather than a single token.

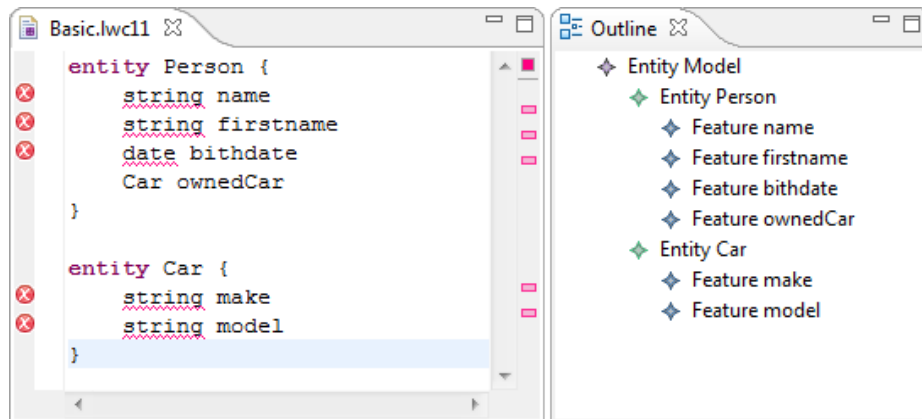


Fig. 4. Example entity model (types not resolving yet)

After generating the DSL tooling from this `.cs` specification by invoking a right-click on the `lwc11.cs` file and selecting **Generate Text Resource**, we're ready to create textual entity models. Figure 4 shows an example model. You can create such models by firing up a second Eclipse instance and creating a new `lwc11` file using `File -> New -> EMFText .lwc11 file`.

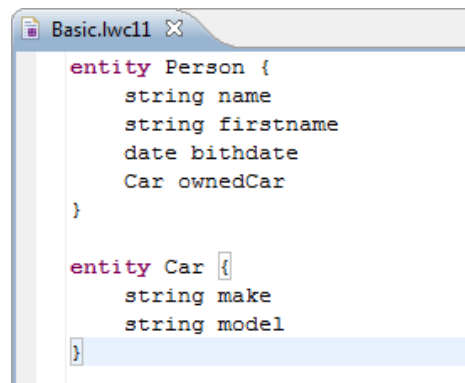
One can see that the primitive types used in Fig. 4 (e.g., `string` and `date`) cannot be resolved yet. The complex types (i.e., other entities) can be resolved using the default reference resolving which is generated by `EMFText`. For example, the type of reference `Person.ownedCar` is correctly resolved to refer to the entity `Car` which is also defined in the example file.

To resolve primitive types correctly, we can create a library for primitive types. We did so by performing a right-click on EClass `TypeLib` and selecting `Create Dynamic Instance` in the sample Ecore tree editor. We created a simple library containing three primitive types (`string`, `date` and `int`) and stored it in `modellib/primitivetypes.xmi`. One can also define textual syntax for primitive types, but we choose to use the XMI representation to show that DSLs generated by EMFText do smoothly integrate with other EMF-based resources.

Now, to resolve references to primitive types correctly, we need to load the created library (i.e., `primitivetypes.xmi`) and look up types by their name, whenever resolving types is requested. To ease this task, EMFText has generated a class `FeatureTypeReferenceResolver` which resides in the `src` folder of the `...lwc11.resource.lwc11` plug-in. The generated implementation of this class delegates all calls to a default reference resolver.

We need to change the method `resolve()` which is called whenever a non-containment reference needs to be resolved. To find the primitive types, we load the `primitivetypes.xmi` resource, search for a type with the correct name and add a mapping for this type if the name matches the identifier that must be resolved. For details on this implementation you may want to have a closer look at the `FeatureTypeReferenceResolver` class.

After implementing this resolution algorithm which is about 20 lines of Java code, the primitive types can be found and the editor shows the same Entity DSL model without any errors (see Fig. 5).



```
entity Person {
    string name
    string firstname
    date bithdate
    Car ownedCar
}

entity Car {
    string make
    string model
}
```

**Fig. 5.** Example entity model (types resolved now)

EMFText itself does not provide dedicated support to customize reference resolving. You can either stick with the default resolving mechanism, which is already quite smart, or implement your own resolution strategy in Java. To use

more sophisticated techniques for reference resolution you may want to consider using JastEMF<sup>5</sup> or EMF Attribute Grammars<sup>6</sup>.

## 2.2 Phase 0.2 Code generation to GPL

To obtain code for a General Purpose Language such a Java or C# for the Entity DSL models, there is two possibilities. First, you can use a Model-2-Text tool of your choice (e.g., JET<sup>7</sup>, Acceleo<sup>8</sup> or Xpand<sup>9</sup>). EMFText itself is specifically built to map between models and text that represent *the same* language. If you want to transform the Entity models to another language you're better of using one of the tools listed above. In particular the tools that are based on EMF should be able to load Entity models without any problems. There is no need to transform the textual models to XMI as EMF takes care of using the correct plug-in to load the models from their textual representation.

Second, if you're lucky and there is a metamodel and a textual syntax of your GPL available, you can use a Model-2-Model transformation instead of a Model-2-Text tool. For the Java language, a metamodel and syntax is available from the JaMoPP homepage<sup>10</sup>. Thus, you can use an arbitrary M2M tool (e.g., ATL or QVTO<sup>11</sup>). The advantage of this procedure is that your transformation will produce a correctly structured model of a Java programm instead of plain text.

As code generation is not an objective of EMFText, we did not perform this task for the Entity DSL, but you can find an example on how to generate Java code from UML models with ATL on the JaMoPP homepage<sup>12</sup>.

## 2.3 Phase 0.3 Simple constraint checks

Performing constraint checks on the Entity DSL models can be achieved in different ways. First, EMFText integrates with the EMF Validation Framework. Thus, any registered validator for your metamodel will be called by the generated editor and errors will be shown. You can also use dedicated constraint languages (e.g., OCL or EVL<sup>13</sup>).

Second, EMFText allows to register post processors, which are called after models are created from text. Such post processors can be used to check constraints using Java code. Simply put, you can register a class with one of the extension points that is defined by the `...lwc11.resource.lwc11` plug-in, add some code which inspects your models and attaches errors to it if constraints

---

<sup>5</sup> <http://code.google.com/p/jastemf>

<sup>6</sup> [http://emftext.org/index.php/EMFText\\_Concrete\\_Syntax\\_Zoo\\_EAG](http://emftext.org/index.php/EMFText_Concrete_Syntax_Zoo_EAG)

<sup>7</sup> <http://www.eclipse.org/emft/projects/jet>

<sup>8</sup> <http://www.eclipse.org/acceleo>

<sup>9</sup> <http://www.eclipse.org/modeling/m2t/?project=xpand>

<sup>10</sup> <http://www.jamopp.org>

<sup>11</sup> <http://www.eclipse.org/m2m/>

<sup>12</sup> [http://www.jamopp.org/index.php/JaMoPP\\_Applications\\_ATL](http://www.jamopp.org/index.php/JaMoPP_Applications_ATL)

<sup>13</sup> <http://www.eclipse.org/gmt/epsilon/doc/evl>

are violated. You can consult the EMFText User Guide<sup>14</sup> for more detailed information on how to implement such a post processor.

**Listing 1.2.** Post processor that detects duplicate names

```
public class ConstraintChecker implements
    ILwc11OptionProvider,
    ILwc11ResourcePostProcessorProvider,
    ILwc11ResourcePostProcessor {

    public void process(Lwc11Resource resource) {
        EList<EObject> contents = resource.getContents();
        for (EObject contentObject : contents) {
            if (contentObject instanceof EntityModel) {
                EntityModel em = (EntityModel) contentObject;
                checkForDuplicateNames(em, resource);
            }
        }
    }

    private void checkForDuplicateNames(
        EntityModel em,
        Lwc11Resource resource) {

        Set<String> usedNames = new LinkedHashSet<String>();
        TreeIterator<EObject> allContents = em.eAllContents();
        while (allContents.hasNext()) {
            EObject next = allContents.next();
            if (next instanceof NamedElement) {
                NamedElement element = (NamedElement) next;
                String name = element.getName();
                if (usedNames.contains(name)) {
                    resource.addError("Found duplicate name.", element);
                } else {
                    usedNames.add(name);
                }
            }
        }
    }

    public ILwc11ResourcePostProcessor getResourcePostProcessor() {
        return this;
    }

    public Map<?, ?> getOptions() {
```

---

<sup>14</sup> [http://www.emftext.org/index.php/EMFText\\_Documentation](http://www.emftext.org/index.php/EMFText_Documentation)

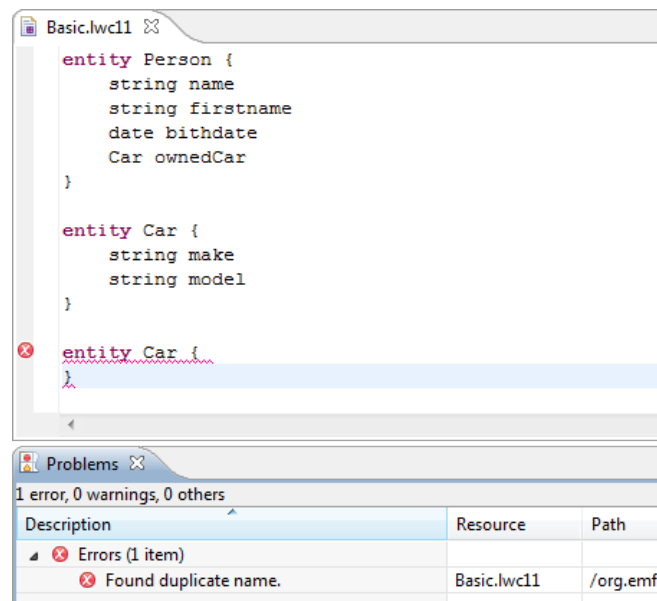


```

return Collections.singletonMap(
    ILwc11Options.RESOURCE_POSTPROCESSOR_PROVIDER,
    this);
}
}

```

To show at least a basic example, consider Listing 1.2 which show a post processor class that check that there are no duplicate names. Once this post processor is correctly registered using the Eclipse extension mechanism, duplicate name will we marked in the editor. The result is shown in Fig. 6.



**Fig. 6.** Duplicate name detection

It is good practice to put the constraint checking classes into separate plug-ins. For example, the class `ConstraintChecker` from Listing 1.2 can be found in the `org.emftext.language.lwc11.constraints` plug-in. In contrast to registering validation classes with the Eclipse Validation Framework, the post processors are only called by EMFText editors and not by other editors for the same metamodel. If you have multiple concrete syntaxes (e.g., a textual and a graphical one) it is better to implement constraint checks based on the Eclipse Validation Framework.

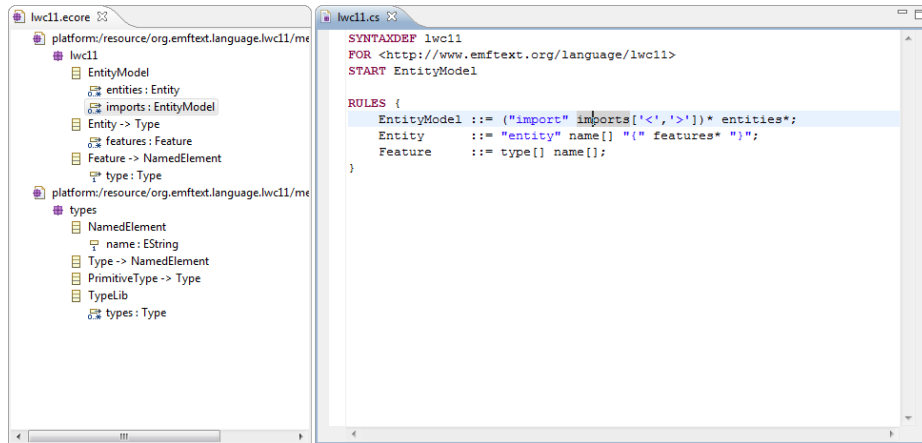


Fig. 7. Metamodel extension and syntax for imports

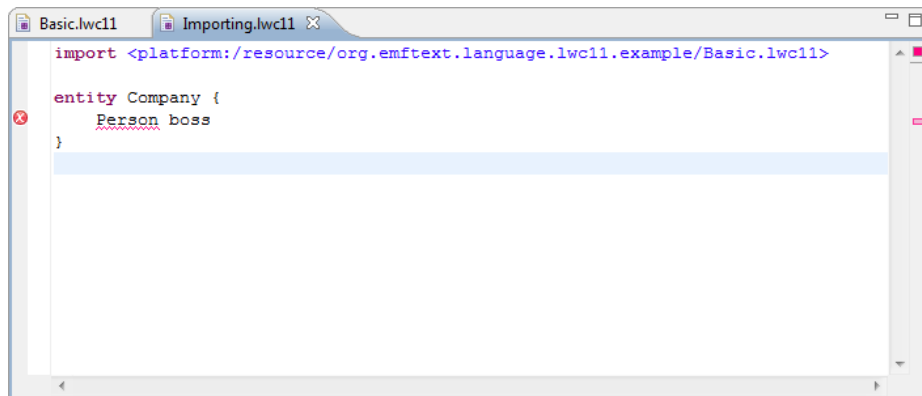


Fig. 8. Example entity model with import (Resolving not working yet)

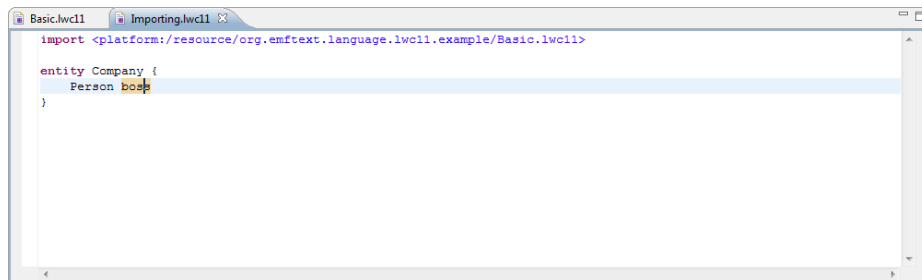


Fig. 9. Example entity model with import (Resolving now working)

## 2.4 Phase 0.4 Splitting models

# 3 Phase 1 - Advanced

## 3.1 Phase 1.1 Language integration

## 3.2 Phase 1.2 Implementing runtime type systems

## 3.3 Phase 1.3 Model-to-model transformation

## 3.4 Phase 1.4 Visibilities, namespaces and scoping for references

## 3.5 Phase 1.5 Integrating manually written code

## 3.6 Phase 1.6 Multiple generators

# 4 Phase 2 - Non-Functional

## 4.1 Phase 2.1 Evolving the DSL without breaking existing models

## 4.2 Phase 2.2 Working with models in a team

## 4.3 Phase 2.3 Scalability of the tools

# 5 Phase 3 - Freestyle

- Use EMFDoc and show documentation in generated editor

## Acknowledgement

**TODO Update.** This research has been co-funded by the European Commission within the 6th Framework Programme project MODELPLEX #034081, the 7th Framework programme project MOST #216691 and by the German Ministry of Education and Research within the project feasiPLe.

## References