Issue

Topic **Model Transformation Framework + Amalthea to Inchron model transformation**

Description

1 Setting up of development environment:

1.1 Code checkout

Source code of APP4MC Model Transformation framework along with examples is available on eclipse repository: https://git.eclipse.org/r/app4mc/org.eclipse.app4mc.tools.git

➔ Use the following command on console to checkout code:

```
Git clone https://git.eclipse.org/r/app4mc/org.eclipse.app4mc.tools.git
```

Branch: **master** is the development branch

➔ Use the following command on console to change branch:

```
Git checkout master
```

1.2 Selection of IDE

- It is recommended to use Eclipse APP4MC product for development, as it already contains Core Eclipse, EMF, Xtend 2, Google Guice plugins.

  Latest version of APP4MC can be downloaded from: https://www.eclipse.org/app4mc/downloads/
- In a case user wants to use "*custom Eclipse IDE*" or "*expect to develop against different target version of Amalthea model*", it is required to use the target file available at the below location.

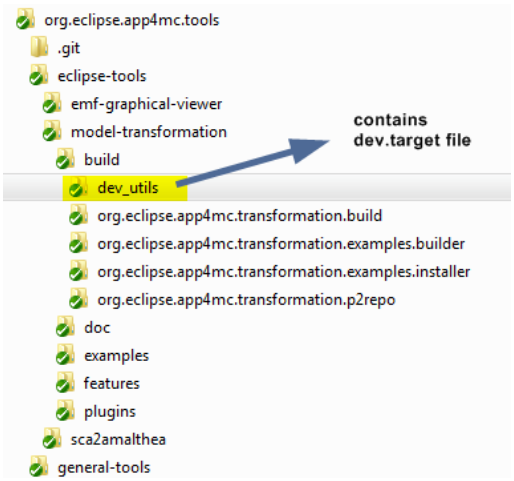


Figure 1 Folder structure of APP4MC tools repo content

Regarding usage of target file and setting up the target for development, have a look at the following chapter

1.3 Importing sources into APP4MC IDE

Follow the steps shown in the screenshot and import the sources into the workspace.

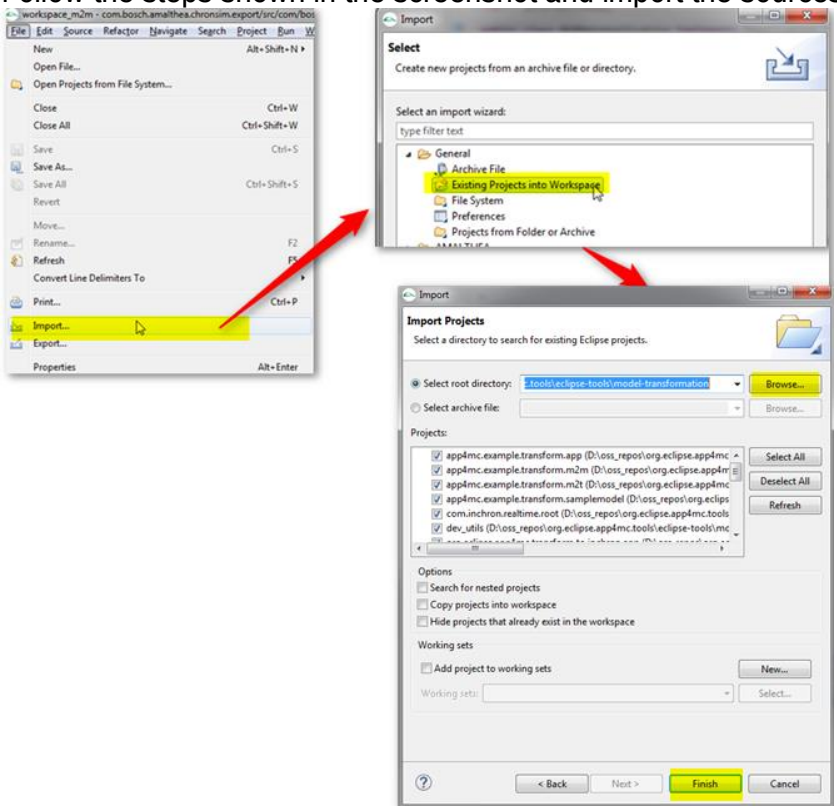


Figure 2 Steps for importing projects into eclipse workspace

1.4 Applying working sets on imported projects

Eclipse Working sets are used to group various projects into custom containers. This is helpful for the developers to organize the projects based on the type.

For organizing projects into working sets, psf file is used.

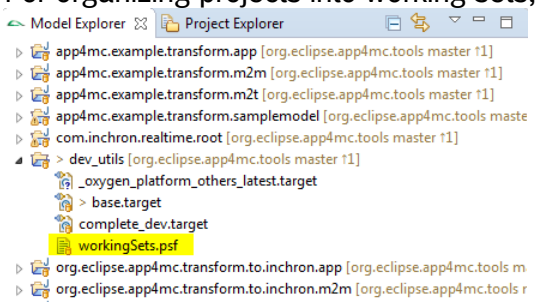


Figure 3 location of working sets psf file

Follow the steps shown in screenshot to use working sets

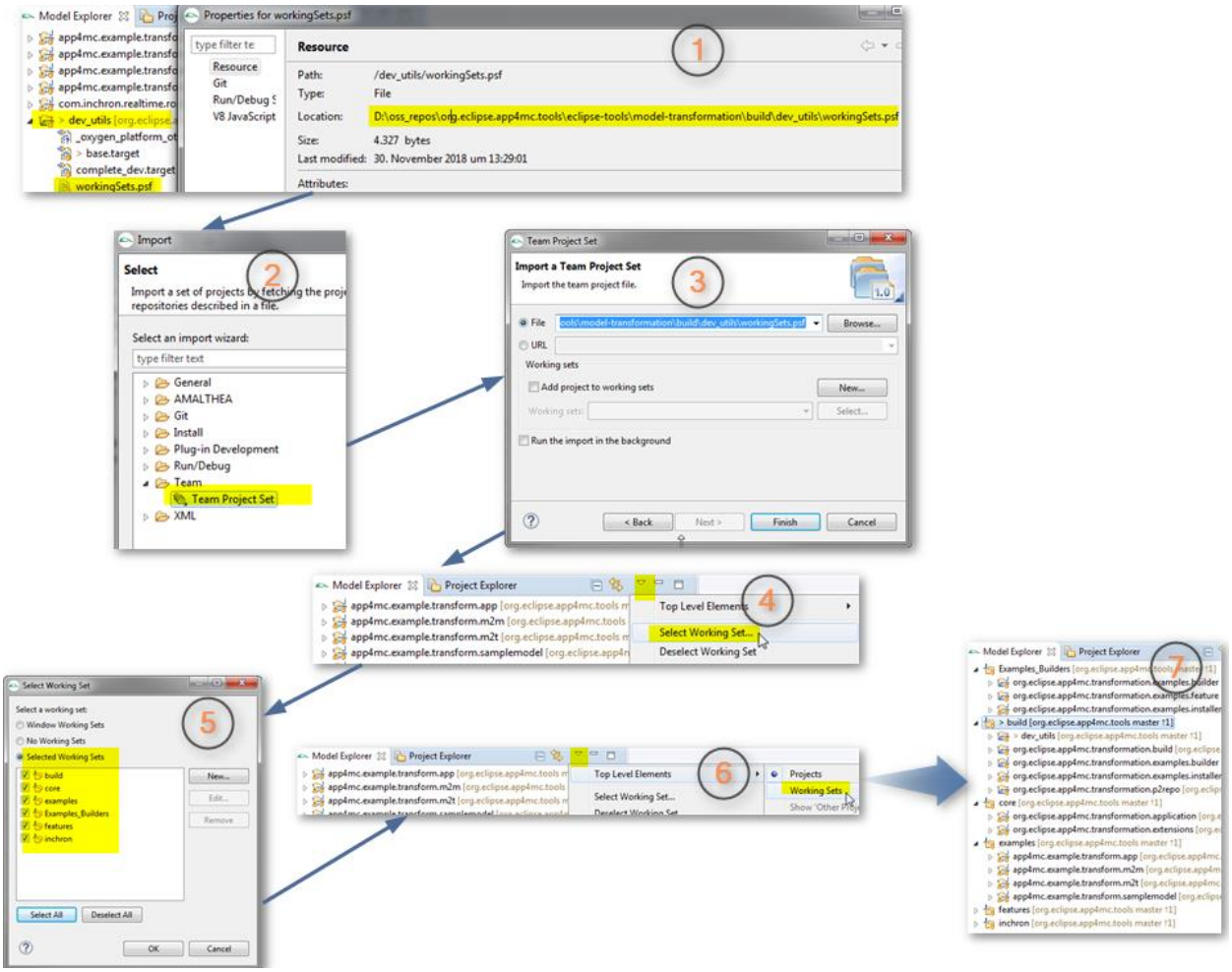


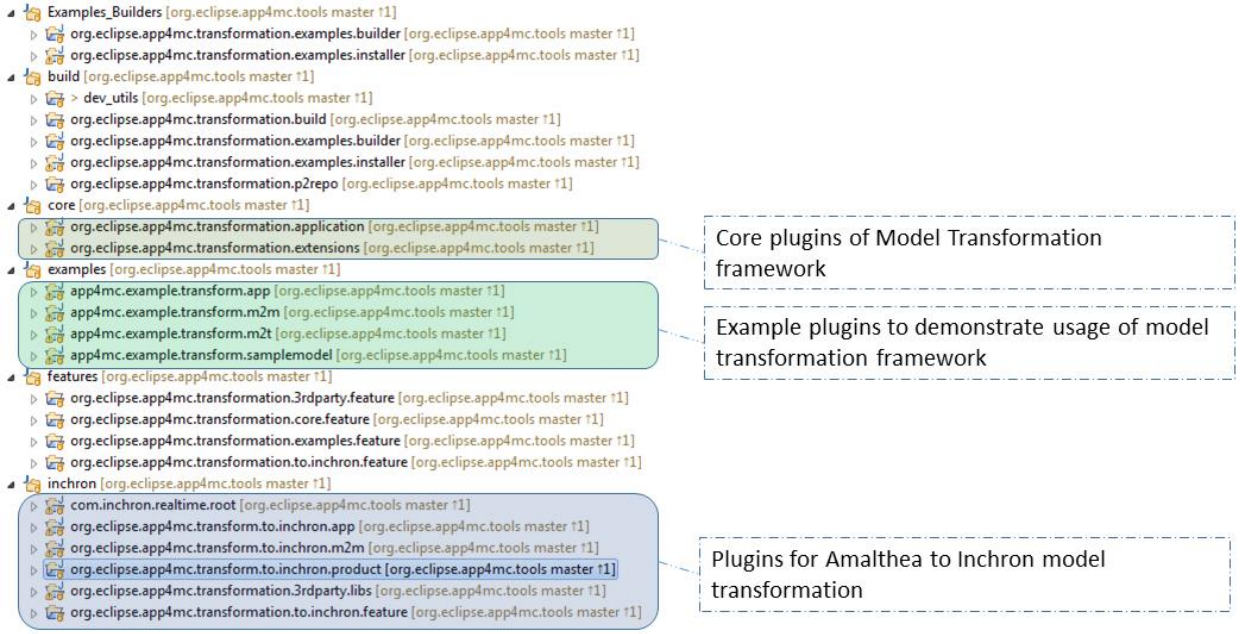
Figure 4 Steps for enabling predefined working sets

2 Use cases included in the current sources

- EMF model transformation framework (“model to model” and “model to text”)
  - Example plugins for Amalthea to a “Simple Model” transformation (*both m2m and m2t*)
- Amalthea to Inchron model transformation (*productive use case: to convert the Amalthea model data to Inchron model*)

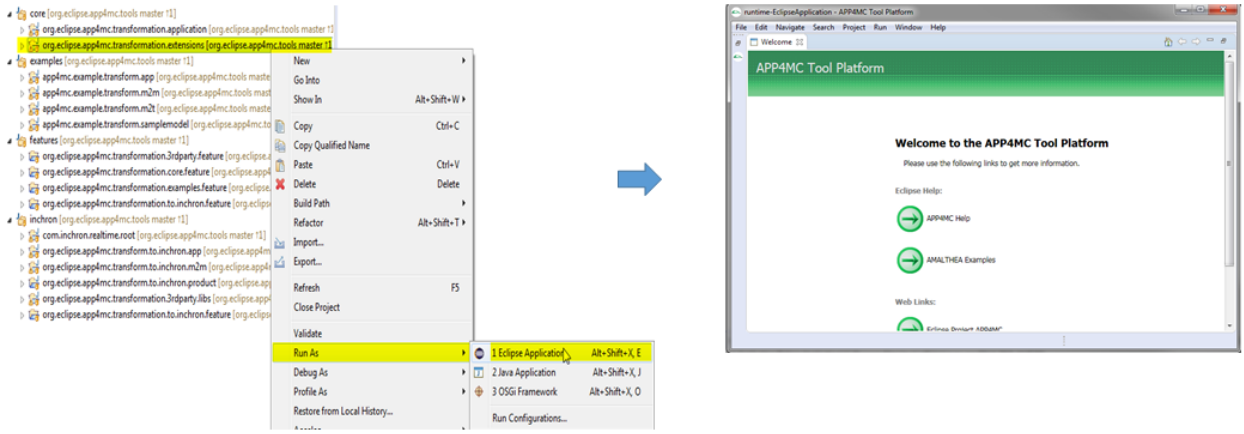
Note: EMF model transformation framework is independent of Amalthea model and it can be used for any model to model/text transformations

3 Source code structure



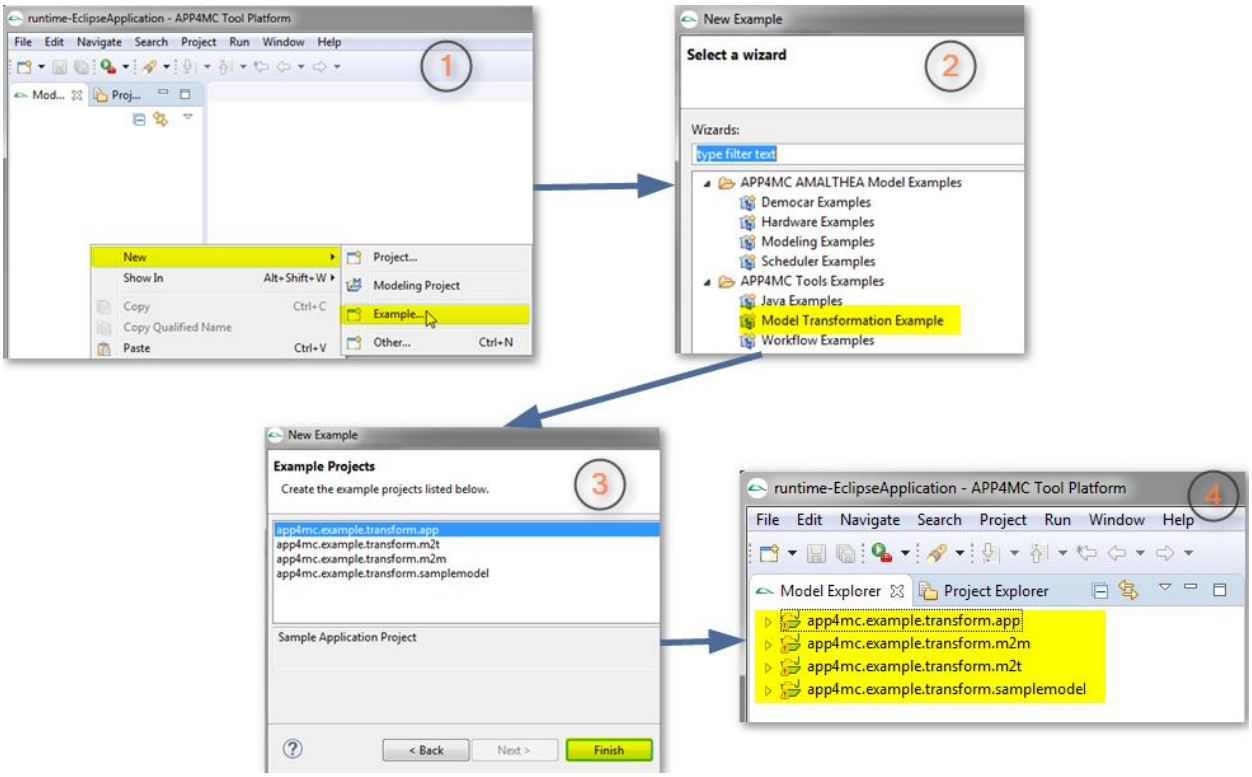
4 Execution of model transformation examples from development environment

4.1 Launch runtime Application



4.2 Create Model Transformation Examples in the runtime environment

Follow the steps as shown in the below screenshot in the runtime environment of eclipse



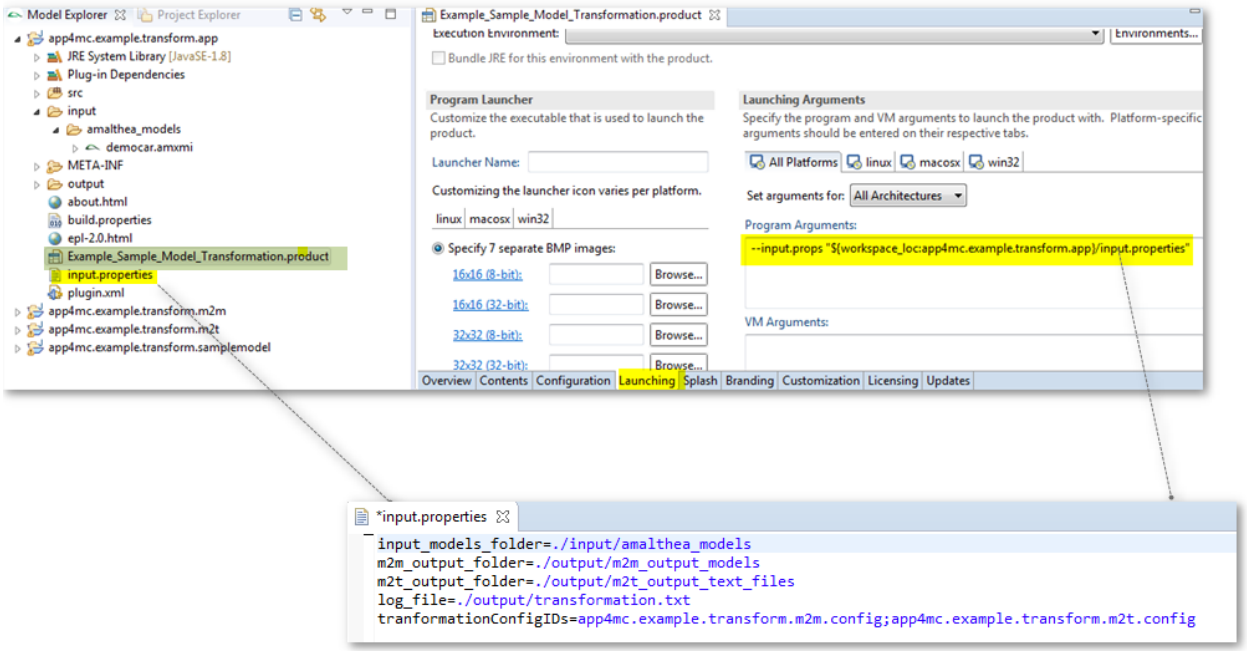
4.3 Input parameters for ”Sample model transformation”:

Input parameters for execution of transformation are specified in the *Example_Sample_Model_Transformation.product* file present in app4mc.example.transform.app plugin.

➔ As shown in the below screenshot, Launching tab of the product file contains location of the input properties file.

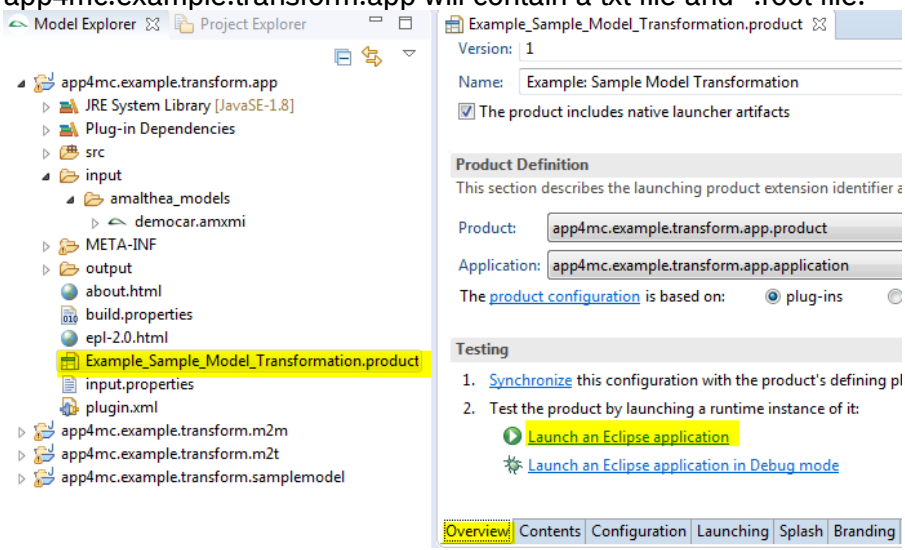
Input properties file should contain below properties in the form of key and value pairs

input_models_folder
m2m_output_folder
m2t_output_folder
log_file
transformationConfigIDs



4.4 Invocation of the model transformation

By clicking on Launch an Eclipse application in the product file (*as shown in screenshot*), Sample model transformation is invoked. On successful execution, output folder in plugin app4mc.example.transform.app will contain a txt file and *.root file.

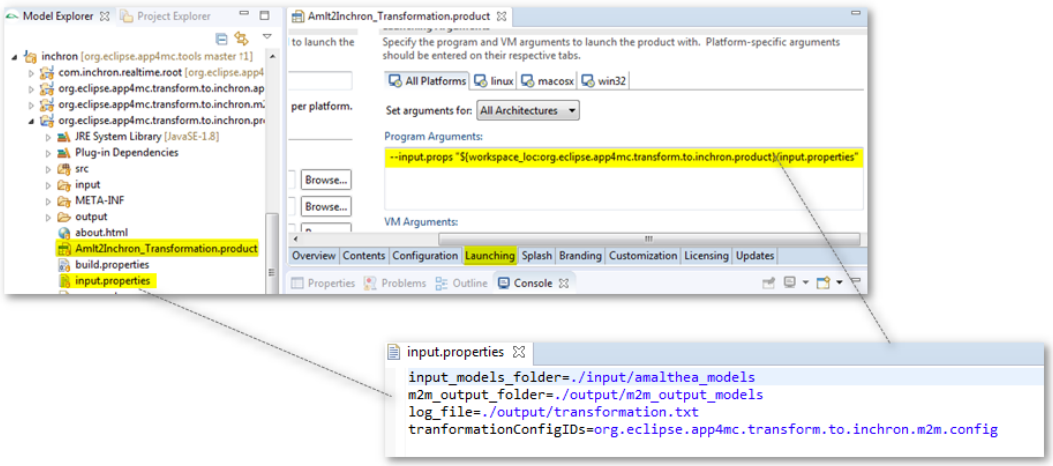


5 Execution of Amalthea to Inchron model transformation

As Amalthea to Inchron model transformation is an individual application which is making use of model transformation framework, it can be executed directly from development workspace without launching runtime application.

5.1 Input parameters to launch the application

Input parameters for execution of model transformation are supplied in file (Amlt2Inchron_Transformation.product) -> launching tab



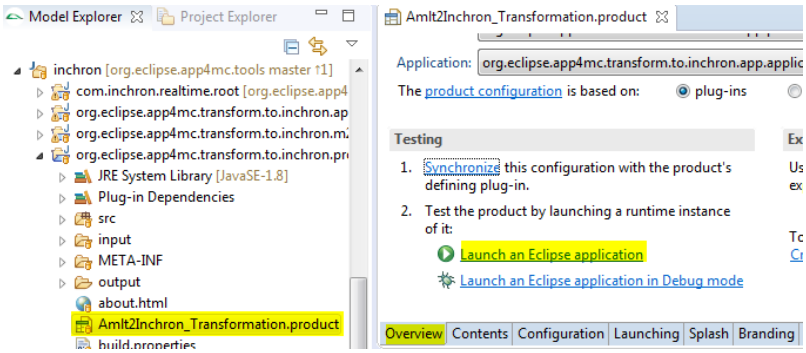
Input properties file should contain below properties in the form of key and value pairs

input_models_folder	Location of Amalthea model files
m2m_output_folder	Location to generate Inchron model file
log_file	Location of log file of model transformation
transformationConfigIDs	Model transformation extension ID (*for Amalthea to Inchron*)

Note: different location of the properties file can be specified in the product file. Also the contents of input.properties can be changed based on the need

5.2 Execution of model transformation

By clicking on Launch an Eclipse application in the product file (*as shown in screenshot*), Amalthea to Inchron model transformation is invoked. On successful execution, output folder in plugin org.eclipse.app4mc.transform.to.inchron.product will contain a txt file and *.root file.



6 Execution from exported product

TBD...

7 Setting runtime platform for model transformation

Target file is used to define the runtime platform. In case runtime platform is not set explicitly, all the plugins available in the dev environment will also be part of the runtime environment.

For Amalthea to Inchron model transformation, as the development is based on fixed Amalthea model version for each release.

- If Amalthea model version required and the APP4MC IDE which is used for development, are of different Amalthea versions -> then it is required to use the dev.target file and specify the required APP4MC release update site

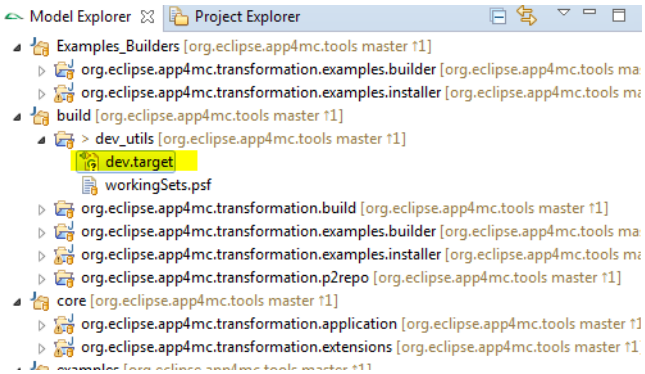


Figure 5 Target file location

Dev.target file present inside dev_utils project, contains the runtime target platform.

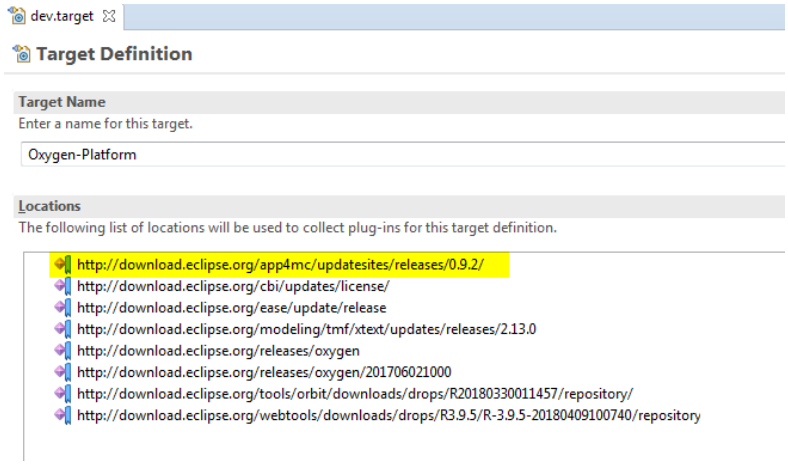
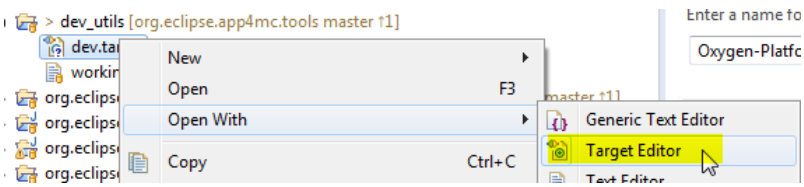


Figure 6 Target file contents

For setting the target platform, open the file with Target editor as shown below and wait till the platform is completely loaded. Once the loading is successful, ensure that there is no message in lower right corner of eclipse, and then press on the "**Set as Active Target Platform**"



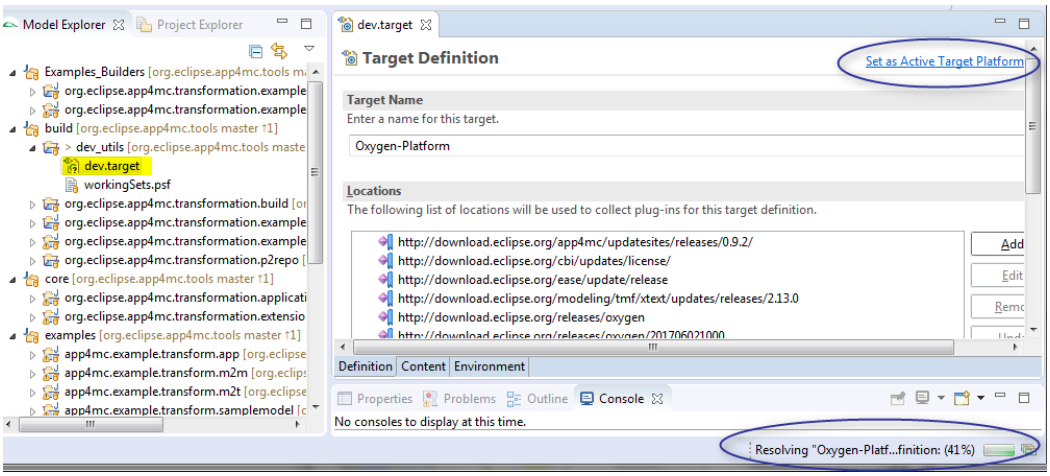


Figure 7 loading of target contents

8 Software structure overview

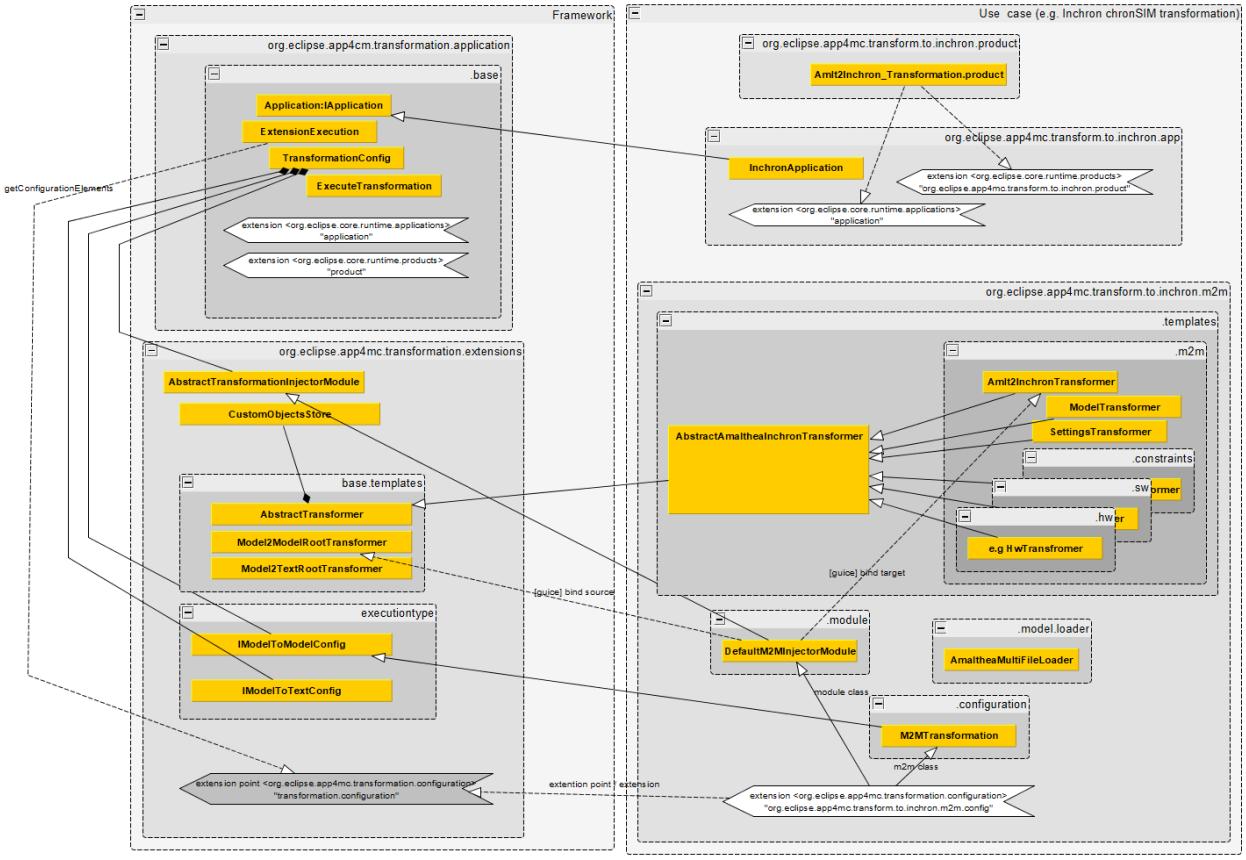


Figure 8: Software structure overview (framework and example Inchron usecase)

8.1 Xtend "create" methods

Xtend's "create" methods are extensively used in the model transformation framework (Inchron chronSIM use case), as they solve the problem of two clause transformations that occurs when a reference to an object is be encountered prior to the object itself. It allows creating an instance based on the parameters supplied to the "create"-method and retrieving the very same instance if calling the same method again with the same parameters, instead of just creating a second instance. Internally this is solved by adding a cache to the method, that keeps track of all created instances and the parameter set they rely on.

For more information see section "Create Methods" within

https://www.eclipse.org/xtend/documentation/202_xtend_classes_members.html

8.2 Google Guice "@Singleton" annotation

Google Guice is a powerful injection framework offering a large number of options to the user. Next to injections and class bindings the Transformation Frameworks makes extensive use of the singleton functionality for its Transformer classes.
As explained in the previous section "create"-methods keep track of the created instances using an internal cache and the supplied method parameters. This cache is essential for the transformation to prevent instantiation of multiple objects from the same input parameter.
However if the embedding class (the transformer class in this case) loses its references, garbage collection might wipe the cache of the "create"-method and causing re-instantiation based on the same parameter set. Therefore one must assure that transformer instance is kept alive and returned, when injecting a transformer at two points in the code, . Adding the @singleton statement makes the class a singleton, which means it is instantiated only upon first call and reused on all subsequent.

9 How to add user defined transformers

9.1 General Info

All transformers in the Amalthea transformation framework may be derived from the abstract AbstractTransformer class, that provides basic infrastructure, as static objects, like a logger, properties, injector (Google Guice) and a CustomObjectsStore. The latter is used to store objects that are created during one transformation for subsequent use in another transformation clause. I also may be used to store side data that is required while running the transformation. In the Inchron chronSIM based example, AbstractTransformer is extended by the abstract AbstractAmaltheaInchronTransformer class that adds some convenience features to the base class, as for example getters for various EMF model factories (Amalthea and Inchron chronSIM). Figure 1 shows the non-exhaustive class hierarchy, as used in the Inchron-flavor transformation framework.

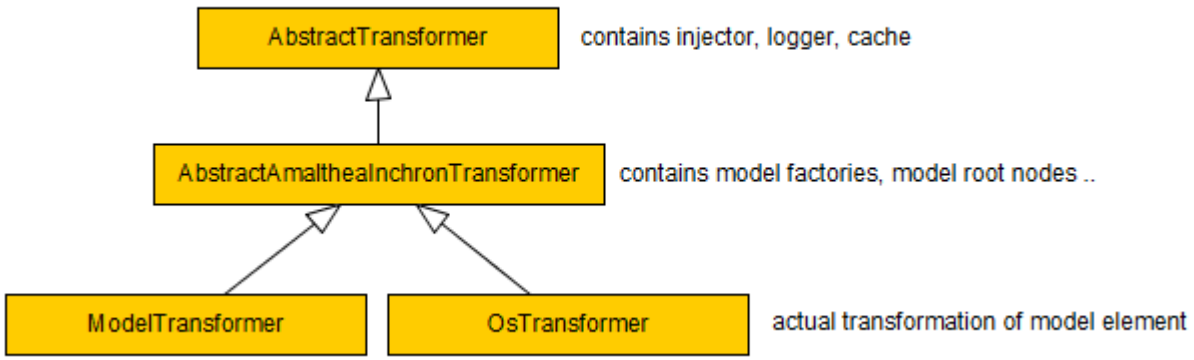


Figure 9: Transformer class hierarchy (non-exhaustive)

9.2 Add a new transformer

It is good practice to derive transformers from the abstract AbstractTransformer class, to make use of its basic infrastructure elements, such as the Guice injector. Thus the introduction of a new transformation element is fairly easy.

1. Add a new transformer class derived from AbstractTransformer, or in case of the chronSIM example from AbstractAmaltheaInchronTransformer
2. Consider making the class a singleton, using the @Singleton statement. This is especially useful if xtend's "create"-methods are used (see subsections Xtend: "create"-methods and Google Guice: @Singleton)
3. Inject the Transformer at any point reachable for the injector, by adding the @Inject keyword preceeding the instance declaration, e.g.:

```
@Inject FrequencyDomainTransformer frequencyDomainTransformer
```

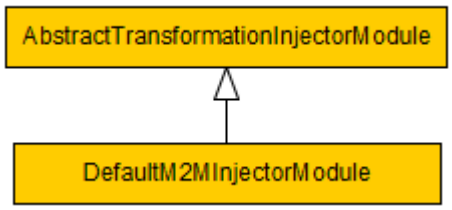


Figure 10: Exemplary injection module (Inchron chronSIM use case)

9.3 Extend/replace an existing transformer

In case an existing transformer's functionality must be extended , or the transformer is to be replaced altogether with a custom implementation, the integration of Google Guice injection mechanisms into the Transformation Framework provide great flexibility and minimal intrusiveness.

Thus the abstract framework class AbstractTransformationInjectorModule provides an interface to hook custom transformers into the flow. Reference 1 from the chronSIM example illustrates a problem, where FrequencyDomainTransformer is used within the CacheTransformer class. Here an instance of FrequencyDomainTransformer is instantiated as a singleton, as specified by the @Inject keyword, preceding the class declaration of FrequencyTransformer. By default the class with the matching name is injected, but this mapping can be adjusted manually in the DefaultM2MInjectorModule class implementing the abstract initialization methods from AbstractTransformationInjectorModuls (see Figure2).

To globally replace `FrequencyTransformer` with `UserDefined_FrequencyTransformer` in all transformation clauses, add the following statement to override the default mapping:

```
bind(FrequencyTransformer.class)
    .to(UserDefined_FrequencyTransformer.class)
```

Figure 11: Bind user-defined transformer with Google Guice

A number of statements to override default injection for all existing Transformers is provided in the comment section within DefaultM2MInjectorModule.

```
import com.google.inject.Inject
import com.inchron.realtime.root.model.memory.MemoryType
import org.eclipse.app4mc.amalthea.model.Cache
import templates.AbstractAmaltheaInchronTransformer
import com.google.inject.Singleton

@Singleton
class CacheTransformer extends AstractAmaltheaInchronTransformer {

     @Inject FrequencyDomainTransformer frequencyDomainTransformer

     def create inchronMemoryFactory.createMemory createCache(Cache
     amltCache) {
          it.name = amltCache.name
          it.clock =
          frequencyDomainTransformer.createClock(amltCache.frequencyDomain)
     }
}

@Singleton
class FrequencyTransformer extends AbstractAmaltheaInchronTransformer {

     def create inchronModelFactory.createFrequency createFrequency(
       Frequency amltFrequency) {
          it.value =
            if(amltFrequency !== null) amltFrequency.value.floatValue else 0
          it.unit = FrequencyUnit.getByName(amltFrequency?.unit.getName)
     }
}

@Singleton
class UserDefined_FrequencyTransformer extends FrequencyTransformer {

     override create inchronModelFactory.createFrequency createFrequency(
       Frequency amltFrequency) {
          it.value = 42
          it.unit = FrequencyUnit.GHZ
     }
}
```

Figure 12: Transfomer override code example