Getting Started with VIATRA

Table of Contents

1. Introduction 1
2. Setting up the tutorial 2
3. Query Development 5
    3.1. Define your First Query 5
    3.2. Query Explorer 5
    3.3. Pattern Language 6
    3.4. Validation 8
    3.5. Advanced Queries 9
    3.6. References 9
4. Model Transformation Development 11
    4.1. Batch Transformations 11
    4.2. Event-driven Transformations 23

1. Introduction

VIATRA is an open source model transformation framework, focusing on efficient evaluation of model queries and supports various transformation workflows. This document contains a tutorial for getting started with the query and transformation development.

The tutorial relies on the CPS Demonstrator application from https://github.com/¬viatra/¬viatra-docs/¬blob/¬master/¬cps/¬Home.adoc. The CPS Demonstrator is specified to cover a usual workflow in Model-driven Engineering, where a system is (1) first described in a source model, then (2) automated model-to-model transformations are used to derive a target model. Finally, (3) model-to-text transformation is performed to generate code from the target domain. In addition, a model generator that can automatically create source domain models can support the correctness testing and performance evaluation of the components. This tutorial uses only a subset of the transformation problem, as its main goal is to illustrate the basics of the VIATRA environment. Consult the original demonstrator for a more complex transformation example.

We expect readers to be familiar with the basics of Eclipse plug-in development, more specifically creating plug-in projects and defining basic UI extensions such as commands. Additionally, we expect a basic understanding of EMF-based modeling If required, look at the Plug-in development tutorial at http://www.vogella.com/¬tutorials/¬EclipsePlugIn/¬article.html or the EMF tutorial at http://www.vogella.com/¬tutorials/¬EclipseEMF/¬article.html for a short tutorial on these subjects.

This document starts with a quick setup guide, then describes model query development. Both batch and event-driven transformations are covered later.

The source of this tutorial is written in AsciiDoc and is available at https://github.com/¬viatra/¬viatra-docs Contributions are welcome! If you find any mistakes or outdated informations, please open an issue.

This document is intended only as a short tutorial for the usage of the Query and Transformation capabilities of VIATRA. A more detailed documentation is maintained at http://wiki.eclipse.org/¬VIATRA.

2. Setting up the tutorial

The VIATRA website describes where to download VIATRA from: https://www.eclipse.org/¬viatra/¬downloads.php This tutorial was written for version 1.4 (released on 30th September 2016).

VIATRA is available from the p2 repository at http://download.eclipse.org/¬viatra/¬updates/¬release

To install VIATRA, select the *Help/Install New Software...*, then paste the copied URL to the *Work with* field, than press *Enter*. When the view is updated, select the *VIATRA Query and Transformation SDK*. Tick the *Contact all update sites during install...* field. Press *Next*, then *Next* and finally *Finish*. After the install process, you should restart Eclipse.

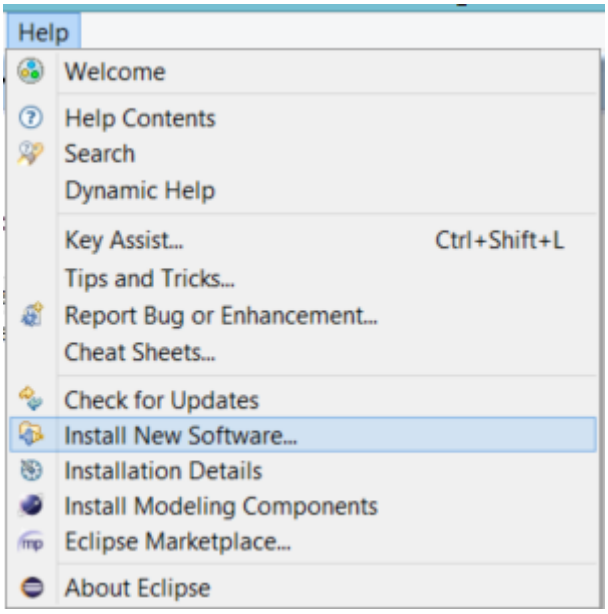

Figure 1. Select **Help/Install New Software...**

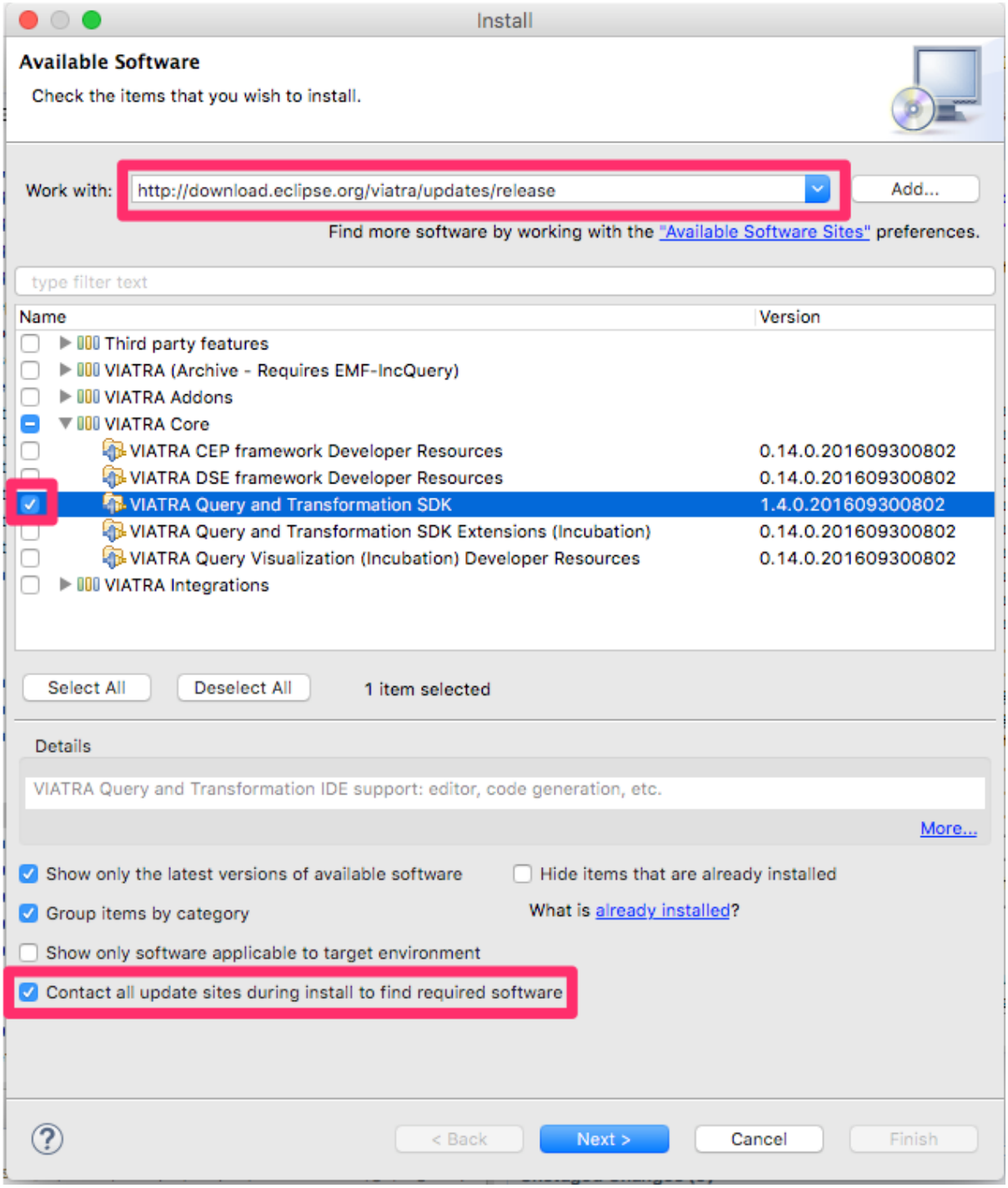

Figure 2. Install window

For a faster installation, you may deselect the *Contact all update sites during install...* field, but then it is necessary to install the required *Xtext SDK* manually.

This tutorial relies on the CPS demonstrator example from https://git.eclipse.org/¬c/¬viatra/¬org.eclipse.viatra.examples.git/¬tree/¬cps (for a description of the example please consult https://github.com/¬viatra/¬viatra-docs/¬blob/¬master/¬cps/¬Home.adoc).

This document assumes that the metamodels themselves are installed into the Eclipse

instance. They are available from the p2 repository https://build.incquerylabs.com/¬jenkins/¬job/¬CPS-Demonstrator/¬lastSuccessfulBuild/¬artifact/¬cps/¬releng/¬org.eclipse.viatra.examples.cps.update/¬target/¬repository/

3. Query Development

3.1. Define your First Query

1. Create a new VIATRA Query project in the host Eclipse with the following name: `com.incquerylabs.cps.queries`.
2. Add `org.eclipse.viatra.examples.cps.model` to the Plug-in dependencies
3. Create a new query definition in a package named `com.incquerylabs.cps.queries` and a file named `queries.vql`. In the wizard create an empty query. Fill the first query:

   queries.vql

   ```
   package com.incquerylabs.cps.queries

   import "http://org.eclipse.viatra/model/cps"

   pattern hostIpAddress(host: HostInstance, ip) {
       HostInstance.nodeIp(host,ip);
   }
   ```

4. Import instances project `models\org.eclipse.viatra.examples.cps.instances` from the CPS Git repository. Note that this is **not** a Git repository URL, but a web interface; a repository url can be found in the bottom of the page.

3.2. Query Explorer

Query Explorer is the primary debug tool for debugging VIATRA Query patterns runtime. Open the view by selecting *Window/Show View/Others/VIATRA/Query Explorer* or you can simply press the `CTRL+3` shortcut and start to type the name of the view. On the left side of the view, there will be patterns inherited from the host eclipse. The middle part will show you the matches of the patterns. To achieve this, we have to load a model into the view:

1. Open our example instance model (/org.eclipse.viatra.examples.cps.instances/example.cyberphysicalsystem)
2. Make sure "ReteEngine" is selected in the Query Explorer (next to the green arrow button)
3. then press the green arrow button on the view.
4. Open the query specification (vql file)

5. then press the green arrow button again

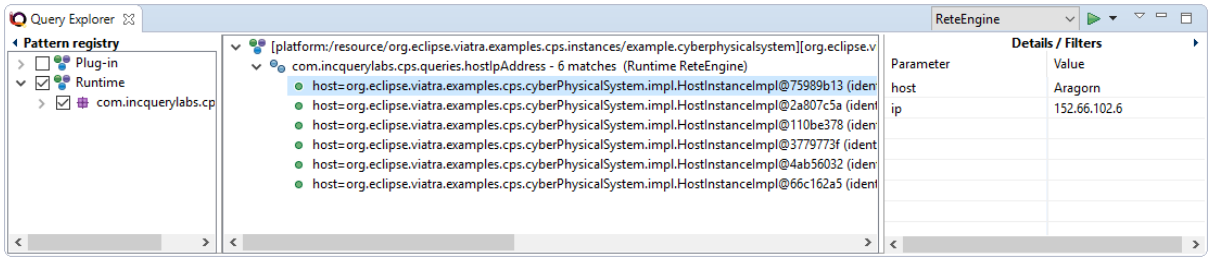

Figure 3. Query Explorer

3.3. Pattern Language

Every pattern has a unique name and several parameters. Inside the body of the patterns, there are different constraints. Our first example describes a feature constraint. It states that `entity` variable is of class `Entity` and its `name` attribute is the value of `name` variable.

1. Create a validation query that checks if the IP address of a `HostInstance` is only an empty string:

```
pattern emptyIpAddress(host: HostInstance) {
    HostInstance.nodeIp(host, "");
}
```

   This pattern shows that the parameters can be typed immediately in the parameters list.

2. Create a validation query that checks if two hosts have the same IP:

```
pattern sameIpAddress(host1, host2, commonIp) {
    HostInstance.nodeIp(host1, commonIp);
    HostInstance.nodeIp(host2, commonIp);
    host1!=host2;
}
```

   This pattern shows the `!=` (*not equal*) operator to select two different entities from the instance model. (Use the `==` operator to equality)

3. Create a validation query that checks if the ip address contains only numbers and dots:

```
pattern ipFormatInvalid(host) {
    HostInstance.nodeIp(host,ip);
    check (
        !ip.matches("^[\\d\\.]+")
    );
}
```

This pattern shows the `check` block where you can write a wide range of *Xbase* expressions (similar to Java). In this case, we define a regular expression. It is important to note that check expressions **have to** be side-effect free and can only be called on attribute variables

4. Create a query that gets the other ending of a relation ending:

```
pattern connectedTo(state: State, other: State){
    State.outgoingTransitions(state, transition);
    Transition.targetState(transition, other);
} or {
    State.outgoingTransitions(other, transition);
    Transition.targetState(transition, state);
}
```

This pattern shows how to connect independent bodies in a pattern by using the `or` keyword that states the pattern has a match if the first *or* the second *or* the third *or* etc body has a match.

5. Create a query that summarizes this three validation condition:

```
pattern badHost(host, ip) {
    find sameIpAddress(host, _other, ip);
} or {
    HostInstance.nodeIp(host, ip);
    find emptyIpAddress(host);
} or {
    HostInstance.nodeIp(host, ip);
    find ipFormatInvalid(host);
}
```

This pattern shows how to reuse previously defined patterns as sub patterns. To do this, we use the `find` keyword then write the id of the sub pattern and finally add the variables. (Variables starting with _ define *don't care* variables, hence you cannot use them in other lines of the pattern)

6. Create a query that matches to the valid hosts:

```
pattern goodEntity(host, ip) {
    HostInstance.nodeIp(host, ip);
    neg find badHost(host, _);
}
```

This pattern shows `neg find` expression to express negation. Those actual parameters of the negative pattern call that are not used elsewhere in the calling body are universally quantified, meaning that the calling pattern only matches if variables of the calling pattern

cannot be bound to matching elements.

7. Create a query that counts the number of attributes of an entity:

```
pattern applications(host, app) {
    HostInstance.applications(host, app);
}

pattern countApplications(host : HostInstance, M) {
    M == count find applications(host, _);
}
```

This pattern shows `count find` expression that aggregates multiple matches of a called pattern into a single value.

3.4. Validation

VIATRA provides facilities to create validation rules based on the pattern language of the framework. These rules can be evaluated on various EMF instance models and upon violations of constraints, markers are automatically created in the Eclipse Problems View.

The **@Constraint** annotation can be used to mark a pattern as a validation rule. If the framework finds at least one pattern with such annotation.

Annotation parameters:

- *key:* The list of paremeters which determine which objects the constraint violation needs to be attached to.
- *message:* The message to display when the constraint violation is found. The message may refer the parameter variables between $ symbols, or their EMF features, such as in $Param1.name$.
- *severity:* "warning" or "error"
- *targetEditorId:* An Eclipse editor ID where the validation framework should register itself to the context menu. Use "*" as a wildcard if the constraint should be used always when validation is started.

To find a specific editor id, we can use the *Plug-in Selection Spy* tool with a `Shift+Alt+F1` shortcut.

For example:

```
@Constraint(targetEditorId =
"org.eclipse.viatra.examples.cps.cyberPhysicalSystem.presentation.Cy
berPhysicalSystemEditorID",
            severity = "error",
            message = "The ip address is not unique",
            key = {"host1"})
pattern sameIpAddress(host1: HostInstance, host2: HostInstance,
commonIp) {
    HostInstance.nodeIp(host1, commonIp);
    HostInstance.nodeIp(host2, commonIp);
    host1!=host2;
}
```

3.5. Advanced Queries

1. Create **Support** pattern:

```
private pattern directReachable(state: State, other: State){
    State.outgoingTransitions(state, transition);
    Transition.targetState(transition, other);
}
```

2. Create a pattern that determines the transitive closure of reachable states:

```
private pattern reachable(state: State, other: State){
    find directReachable+(state, other);
}

pattern reachableState(sm :StateMachine, state: State){
    StateMachine.initial(sm, state);
} or {
    StateMachine.initial(sm, initial);
    StateMachine.states(sm, state);
    find reachable(initial, state);
}
```

3.6. References

- Pattern Language: https://wiki.eclipse.org/¬VIATRA/¬Query/¬UserDocumentation/¬QueryLanguage
- Validation Framework: https://wiki.eclipse.org/¬VIATRA/¬Addons/¬UserDocumentation/¬Validation
- Query Based Features: https://wiki.eclipse.org/¬VIATRA/¬Addons/¬UserDocumentation/¬Query_Based_Featur

es

4. Model Transformation Development

For model transformation development a Java API is available, allowing seamless integration of the transformations into any Java applications. However, to enhance readability, we recommend using a higher-level JVM language, as it allows defining the transformation as an internal DSL of this host language.

In this tutorial we rely on the Xtend language to host the VIATRA transformation DSL, and we rely on its extension method and type inference support to reduce unnecessary elements. However, other JVM-based languages can also be used with similar efficiency (for an example in Kotlin see https://gist.github.com/¬doczir/¬bfe95c470599c5b8e60b400b80f92ea2).

4.1. Batch Transformations

This exercise helps the audience to create a simple batch transformation using the VIATRA Transformation API. The transformation will transform the hosts and applications in a CPS model to a deployment model. The exercise also covers registering a menu command which initializes the transformation.

4.1.1. Create transformation

For the transformation, we have to created a *VIATRA Query Project* (the one from the query development tutorial could also be reused), and create a new query file called *CpsXformM2M.vql* to store the patterns we want to use in the transformation with the following contents:

```
import "http://org.eclipse.viatra/model/cps"
import "http://org.eclipse.viatra/model/deployment"
import "http://org.eclipse.viatra/model/cps-traceability"

pattern hostInstance(hostInstance : HostInstance) {
    HostInstance(hostInstance);
}

pattern applicationInstance(
    appType : ApplicationType,
    appInstance : ApplicationInstance
) {
    HostInstance.applications(_, appInstance);
    ApplicationType.instances(appType, appInstance);
}

/**
 * Traceability link access
 */
pattern cps2depTrace(
    cps2dep : CPSToDeployment,
    trace : CPS2DeploymentTrace,
    cpsElement : Identifiable,
    depElement : DeploymentElement
) {
    CPSToDeployment.traces(cps2dep, trace);
    CPS2DeploymentTrace.cpsElements(trace, cpsElement);
    CPS2DeploymentTrace.deploymentElements(trace, depElement);
}
```

- Create transformation class in Xtend
  - Create new *Model Transformation* with the wizard

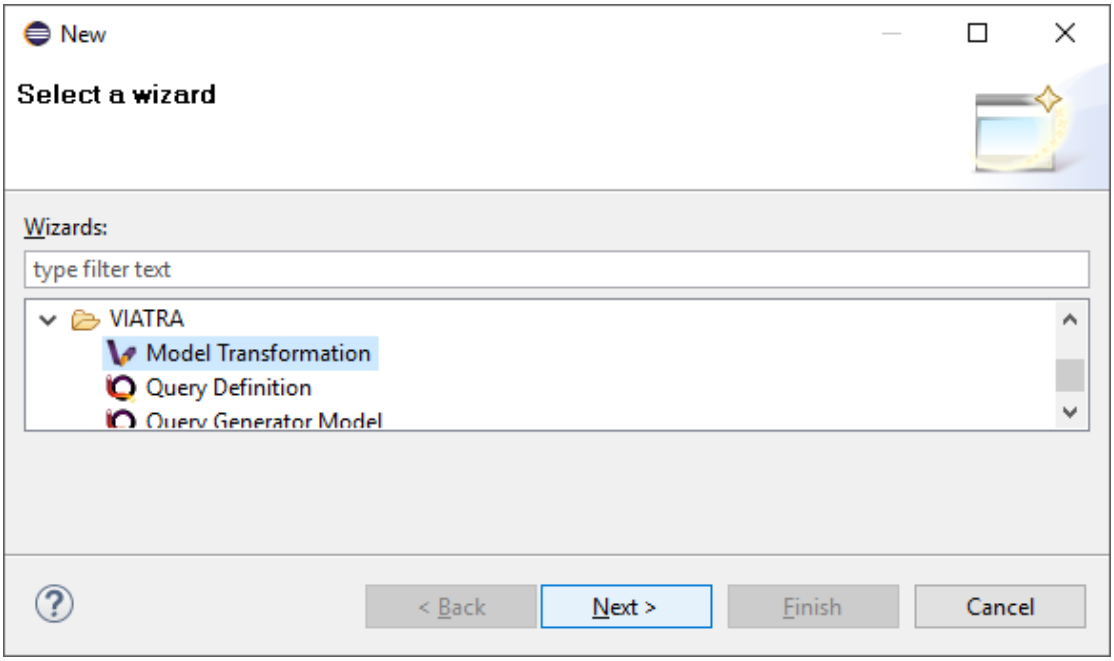

Figure 4. Model Transformation Wizard - Create new transformation

¬ Setup the name of the transformation and click Next

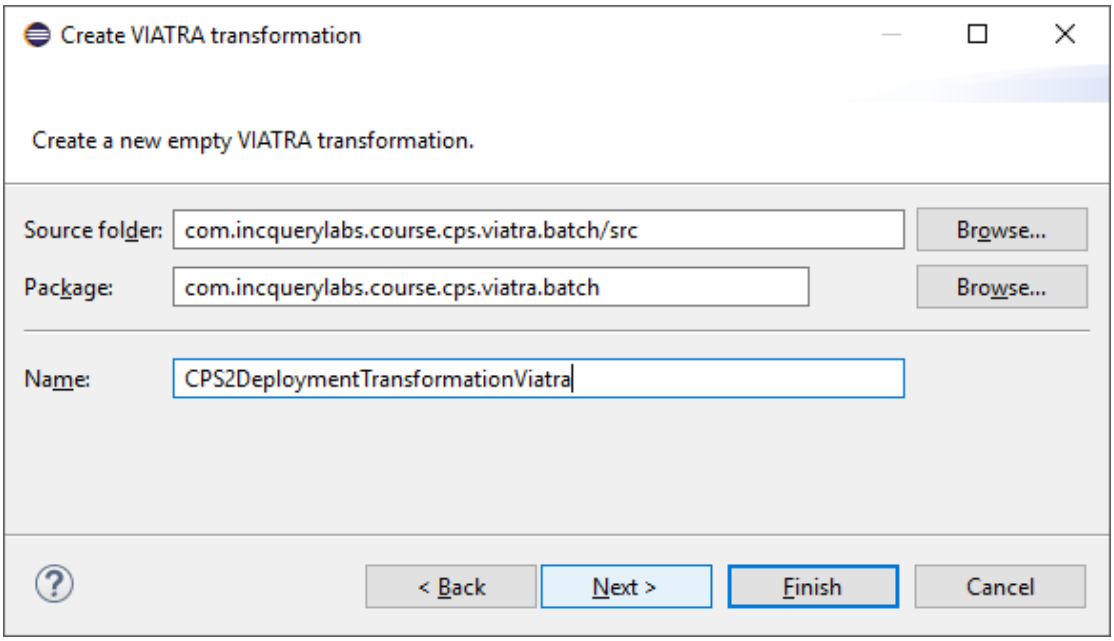

Figure 5. Model Transformation Wizard - Name of the new batch transformation

¬ Setup the type of the transformation to *BatchTransformation* and click Finish

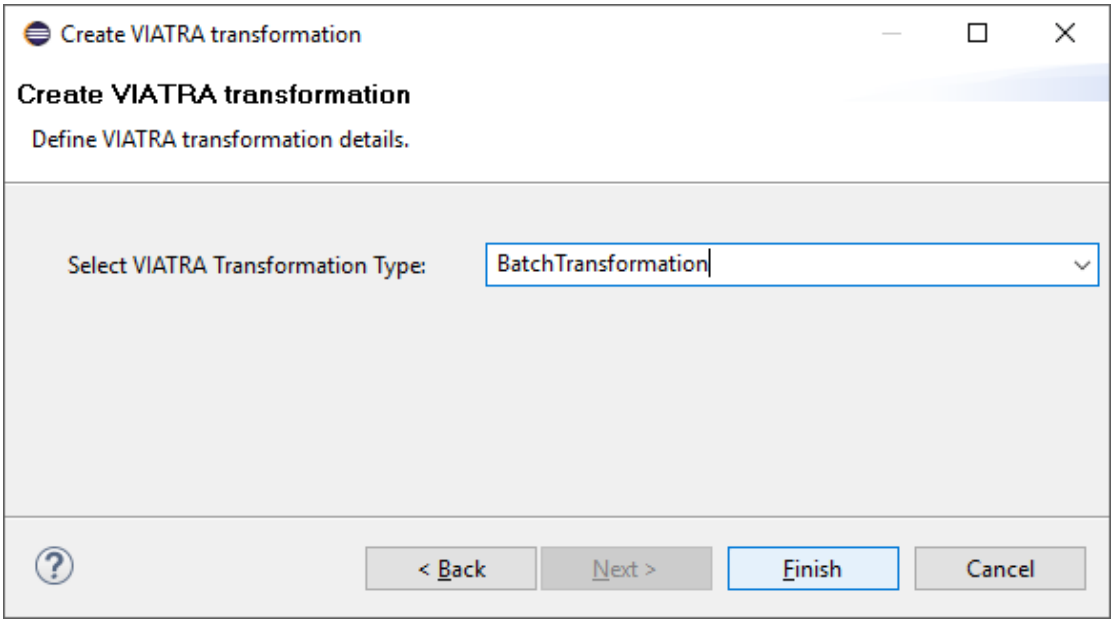

Figure 6. Model Transformation Wizard - Type of the new transformation

In the created file we have to register a few extension methods, more specifically for our used queries (`CpsXformM2m`, the same name the VQL file uses) and the EMF EPackages we want to refer (here the deployment and traceability packages). A few additional extension methods are already registered, e.g. transformation rule builder and model manipulation API.

```
/** VIATRA Query Pattern group **/
val extension CpsXformM2M cpsXformM2M = CpsXformM2M.instance

/** EMF metamodels **/
val extension DeploymentPackage depPackage =
DeploymentPackage.eINSTANCE
val extension TraceabilityPackage trPackage =
TraceabilityPackage.eINSTANCE
```

- Constructor will also initialize transformation (replace the generated one)
  - It assumes that the resource and trace models are already created
  - The IModelManipulations implementation is used to make model access replaceable, this way the same transformation may be used for cases where the resource set is transactional. The initialization of this is generated automatically into the `createTransformation` method.

    ```
    val CPSToDeployment cps2dep

    new(CPSToDeployment cps2dep, ViatraQueryEngine engine) {
        this.cps2dep = cps2dep
        resource = cps2dep.deployment.eResource
        this.engine = engine
        prepare(engine)
        createTransformation
    }
    ```

- Transformation will remain active until disposed is called (there is a generated `dispose` method in the class)
- Create a rule to generate DeploymentHosts for each HostInstances
  - The `BatchTransformationRuleFactory` extension provides a builder API for rule definition
  - A VIATRA query is used as precondition to the rule, which means the rule will be activated each time the given pattern when changes allowing to update the output accordingly.

    ```
    val hostRule =
    createRule.precondition(HostInstanceMatcher.querySpecification)
    .action[/*Action part*/].build
    ```

- Specify which action to run when the rule fires. It will create the transformed `DeploymentHost` element in the output model as well as a trace element associating the source `HostInstance` and the target `DeploymentHost`:

```
val hostRule =
createRule.precondition(HostInstanceMatcher.querySpecification).a
ction[
    val cpsHostInstance = it.hostInstance
    val nodeIp = cpsHostInstance.nodeIp
    println('''Mapping host with IP: «nodeIp»''')

    /** Create & initialize DeploymentHost in output model **/
    val depHost =
cps2dep.deployment.createChild(deployment_Hosts, deploymentHost)
=> [
        set(deploymentHost_Ip, nodeIp)
    ]

    /** Create trace element in trace model **/
    cps2dep.createChild(CPSToDeployment_Traces,
CPS2DeploymentTrace) => [
        addTo(CPS2DeploymentTrace_CpsElements, cpsHostInstance)
        addTo(CPS2DeploymentTrace_DeploymentElements, depHost)
    ]

    println('''Mapped with IP: «nodeIp»''')
].build
```

- The rule which creates `DeploymentApplication` elements for `ApplicationInstance` objects, looks similar. It has to find the `DeploymentHost` created from the `HostInstance` to which the source `ApplicationInstance` is allocated, so it assumes the `hostRule` has already fired:

```
val applicationRule =
createRule.precondition(ApplicationInstanceMatcher.querySpecifica
tion).action[
    val cpsApplicationInstance = it.appInstance
    val appId = cpsApplicationInstance.identifier
    println('''Mapping application with ID: «appId»''')

    /* Find the DeploymentHost created from the HostInstance to
which the source ApplicationInstance is allocated */
    val cpsHostInstance = cpsApplicationInstance.allocatedTo
    val depHost =
engine.cps2depTrace.getAllValuesOfdepElement(null, null,
cpsHostInstance).filter(DeploymentHost).head
    /* Create & initialize DeploymentApplication in this
DeploymentHost */
    val deploymentApplication =
depHost.createChild(deploymentHost_Applications,
deploymentApplication) => [
        set(deploymentApplication_Id, appId)
    ]

    /* Create trace element in trace model */
    cps2dep.createChild(CPSToDeployment_Traces,
CPS2DeploymentTrace) => [
        addTo(CPS2DeploymentTrace_CpsElements,
cpsApplicationInstance)
        addTo(CPS2DeploymentTrace_DeploymentElements,
deploymentApplication)
    ]

    println('''Mapped application with ID: «appId»''')
].build
```

- Implement the method which performs the transformation using the rules defined above:
  - ¬ Since we are using the non-incremental (the whole model is always retransformed on model changes), the output and trace models are to be cleared before the any rule can fire
  - ¬ Pay attention to fire the rules in the proper order

```
def execute() {
    println('''Executing transformation on: Cyber-physical
system: «cps2dep.cps.identifier»''')
    /* Clear output & trace model for batch transformation**/
    cps2dep.deployment.hosts.clear
    cps2dep.traces.clear
    /* Fire transformation rules**/
    hostRule.fireAllCurrent
    applicationRule.fireAllCurrent
}
```

4.1.2. Create a menu command to execute the transformation

- Create a UI plugin with the following additional dependencies:

```
org.eclipse.ui,
com.incquerylabs.course.cps.viatra.batch;bundle-version="0.1.0",
org.eclipse.viatra.examples.cps.traceability;bundle-
version="0.1.0",
org.eclipse.viatra.query.runtime;bundle-version="1.2.0"
```

- Create handler implementation:

  TransformHandler.java

```
public class TransformHandler extends AbstractHandler implements
IHandler {

    ViatraQueryEngine engine;
    CPS2DeploymentTransformationViatra transformation;

    @Override
    public Object execute(ExecutionEvent event) throws
ExecutionException {
        IStructuredSelection selection =
            (IStructuredSelection)
HandlerUtil.getCurrentSelection(event);

        CPSToDeployment tracemodel =
            (CPSToDeployment) selection.getFirstElement();

        if (engine == null){
            try {
                engine = ViatraQueryEngine.on(
                            new EMFScope(

tracemodel.eResource().getResourceSet()));
                transformation = new
CPS2DeploymentTransformationViatra(tracemodel,

engine);
            } catch (ViatraQueryException e) {
                throw new ExecutionException(e.getMessage(), e);
            }
        }
        transformation.execute();

        return null;
    }

}
```

- Register handler in the context menu of `CPSToDeployment` elements in `plugin.xml`:

```
<extension point="org.eclipse.ui.commands">
    <command

defaultHandler="com.incquerylabs.course.cps.viatra.batch.ui.Trans
formHandler"
        id="com.incquerylabs.course.cps.viatra.batch.ui.command"
        name="Transform">
    </command>
</extension>
<extension point="org.eclipse.ui.menus">
    <menuContribution allPopups="false"

locationURI="popup:org.eclipse.ui.popup.any?after=additions">
        <command
commandId="com.incquerylabs.course.cps.viatra.batch.ui.command"
                style="push">
            <visibleWhen checkEnabled="false">
                <with variable="selection">
                    <count value="1">
                    </count>
                    <iterate>
                        <adapt
type="org.eclipse.viatra.examples.cps.traceability.CPSToDeploymen
t">
                        </adapt>
                    </iterate>
                </with>
            </visibleWhen>
        </command>
    </menuContribution>
</extension>
```

4.1.3. Execute the transformation

- Launch *Eclipse Application*
- Create a generic resource project
- Copy a `.cyberphysicalsystem` resource in it if you already have one, or create a new *CaberPhysicalSystem Model*

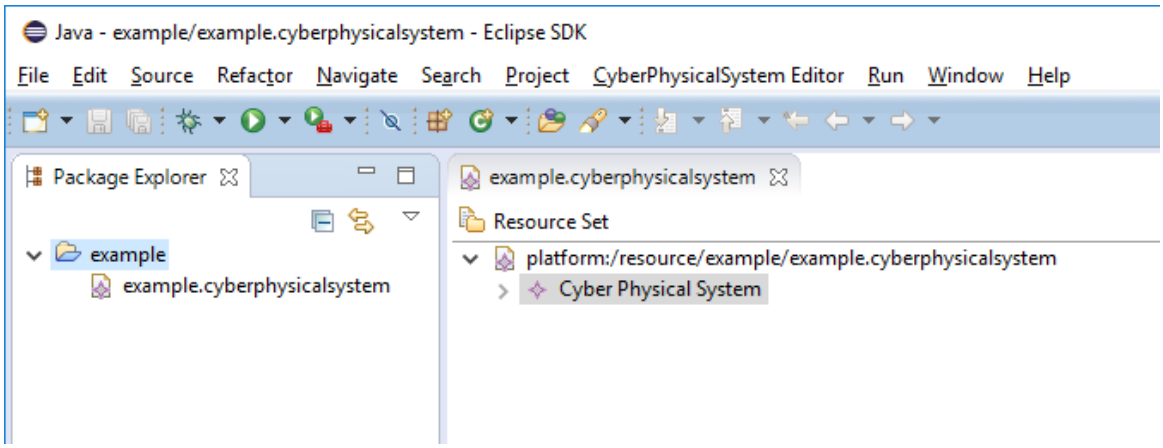

Figure 7. Project with a .cyberphysicalsystem resource

- Create a Deployment model
  - ¬ Root element shall be *Deployment*

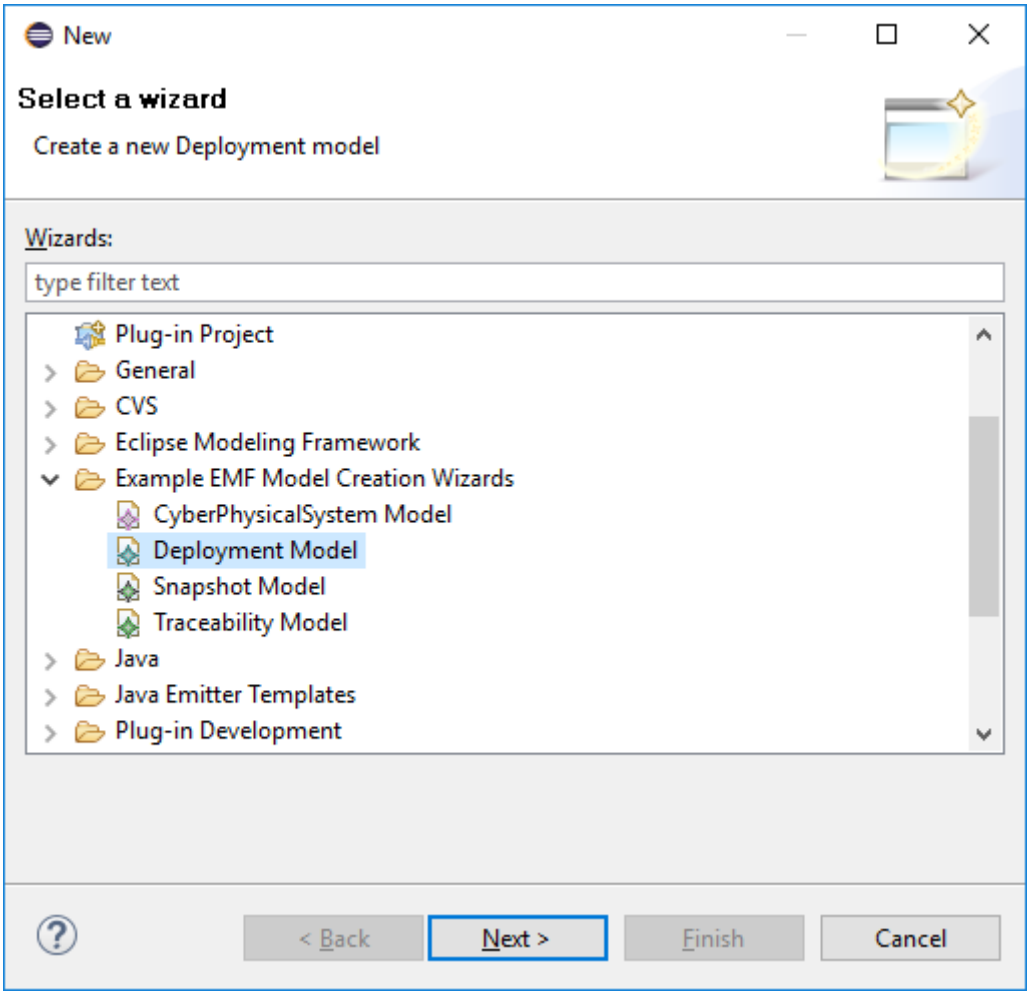

Figure 8. New Deployment Model

- Create a Traceability model
  - ¬ Root element shall be *CPS To Deployment*

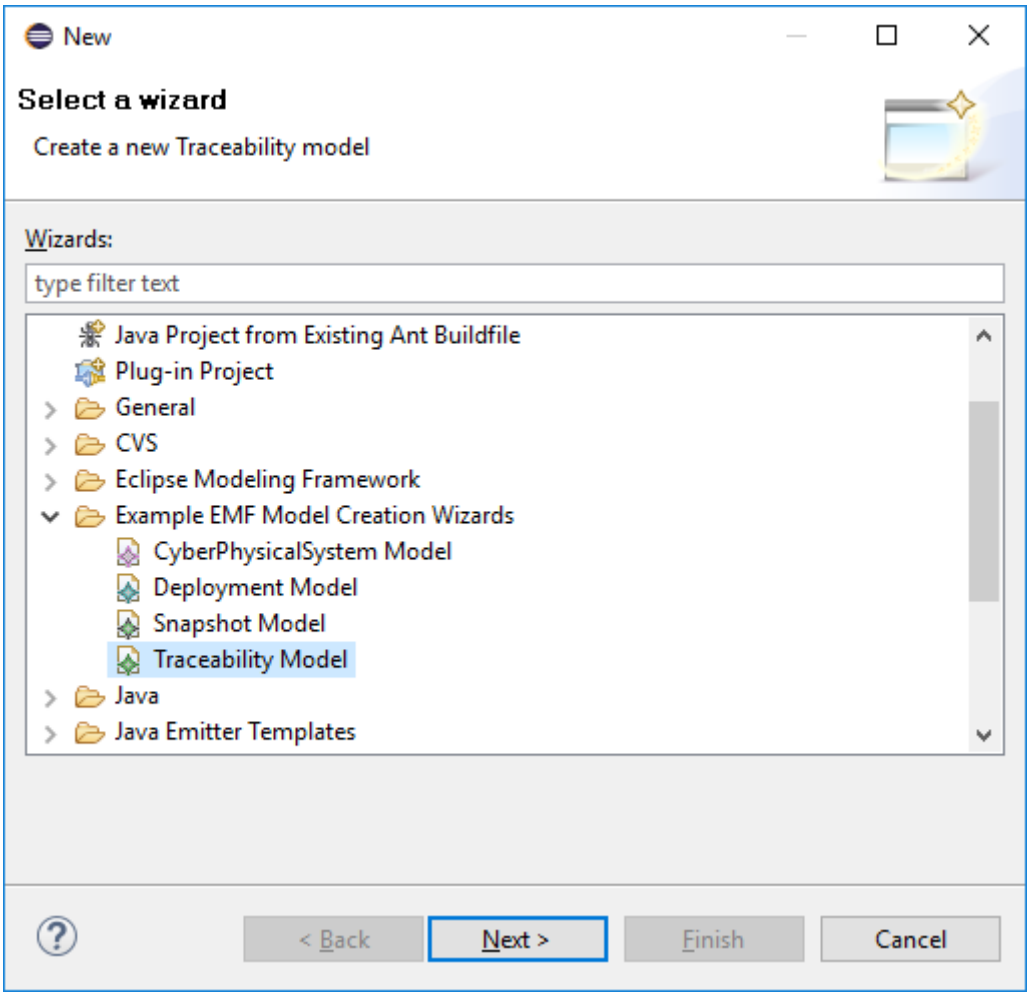

Figure 9. New Traceability Model

- In the Traceability editor, load both CPS and Deployment models with *Load Resources...* in the context menu

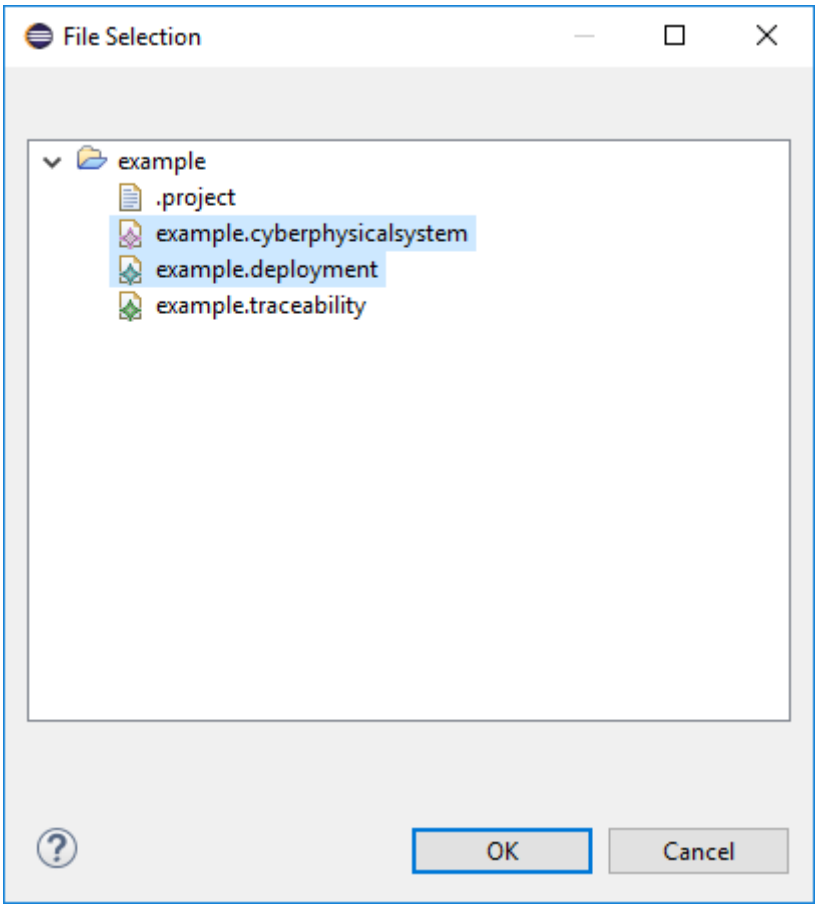

Figure 10. Load necessary resources into the Tracebility Model

- Set CPS and Deployment references of traceability model in the properties view

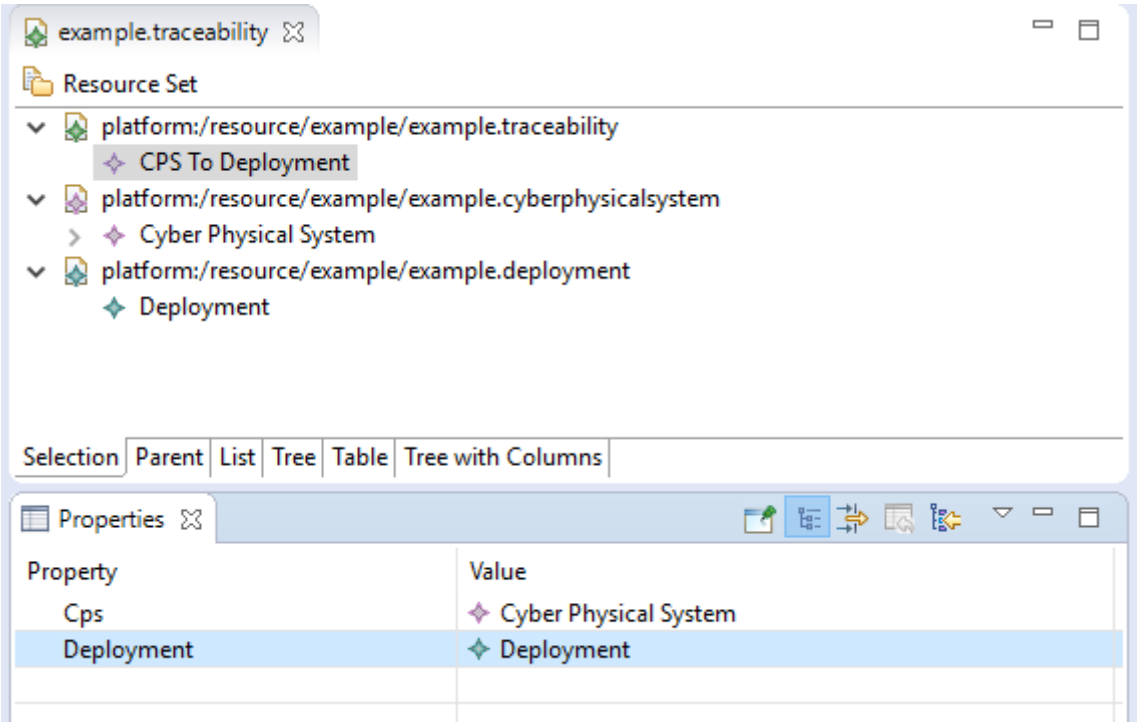

Figure 11. Set the references of the Traceability Model

- Create a new *HostType*, *HostInstance*, *ApplicationType* and *ApplicationInstance* in the

Deployment model

- Execute transformation using the created command (on the context menu of the Traceability model root)

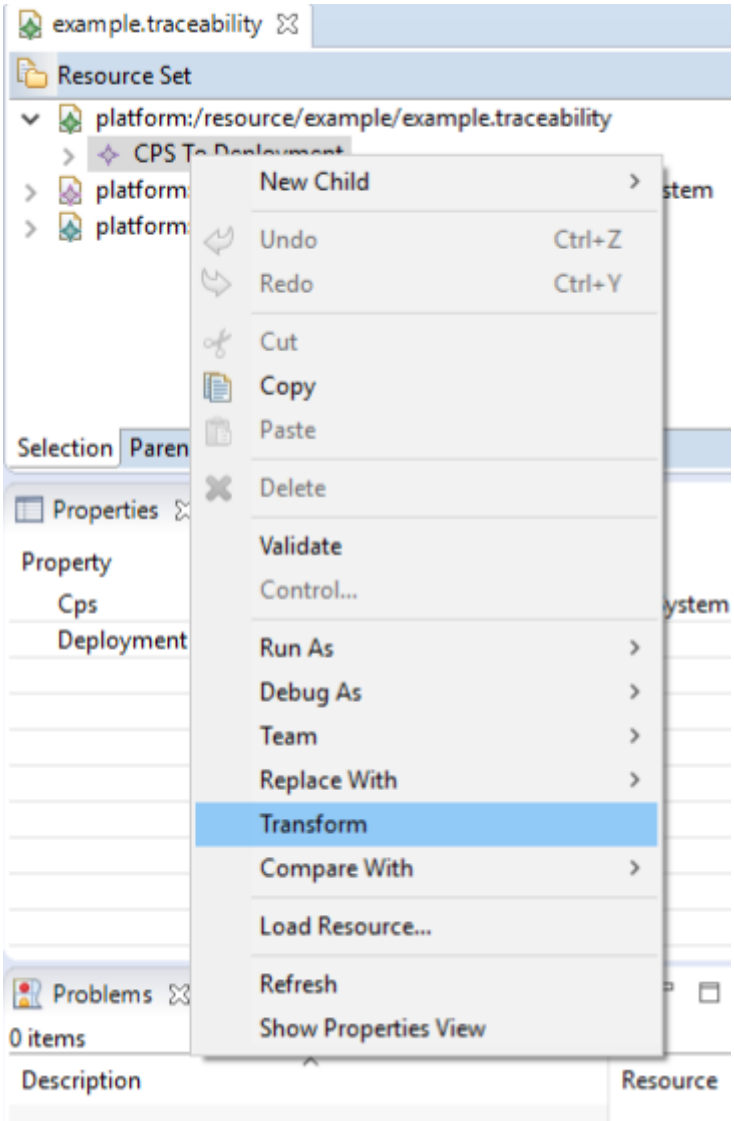

Figure 12. Transformation command in the context menu

4.1.4. References

- VIATRA Transformation API: https://wiki.eclipse.org/¬VIATRA/¬Transformation/¬Transformation_API
- Further transformation examples: https://wiki.eclipse.org/¬VIATRA/¬Transformation/¬Examples

4.2. Event-driven Transformations

This exercise heps the audience to create a simple event-driven transformation using the VIATRA Transformation API. The transformation will create (and then incrementally update while active) a deployment model based on a CPS model. The exercise also covers registering a menu command which initializes the transformation.

Given the batch and event-driven transformations are really similar, this section focuses mainly on the differences; if required, consult the batch transformation tutorial.

4.2.1. Create transformation

Specific patterns have to be defined for event-driven rules; note that there are small differences to the batch definitions, e.g. there is an additional pattern called *allocatedDeploymentApplication*.

```
import "http://org.eclipse.viatra/model/cps"
import "http://org.eclipse.viatra/model/deployment"
import "http://org.eclipse.viatra/model/cps-traceability"

pattern hostInstance(hostInstance) {
    HostInstance(hostInstance);
}

pattern applicationInstance(appType, appInstance){
    HostInstance.applications(_, appInstance);
    ApplicationType.instances(appType, appInstance);
}

pattern allocatedDeploymentApplication(depHost, depApp) {
    DeploymentHost.applications(depHost, depApp);
}

pattern cps2depTrace(cps2dep, trace, cpsElement, depElement) {
    CPSToDeployment.traces(cps2dep, trace);
    CPS2DeploymentTrace.cpsElements(trace, cpsElement);
    CPS2DeploymentTrace.deploymentElements(trace, depElement);
}
```

- Create transformation class (preferably Xtend)
  - ¬ Create new *Model Transformation* with the wizard

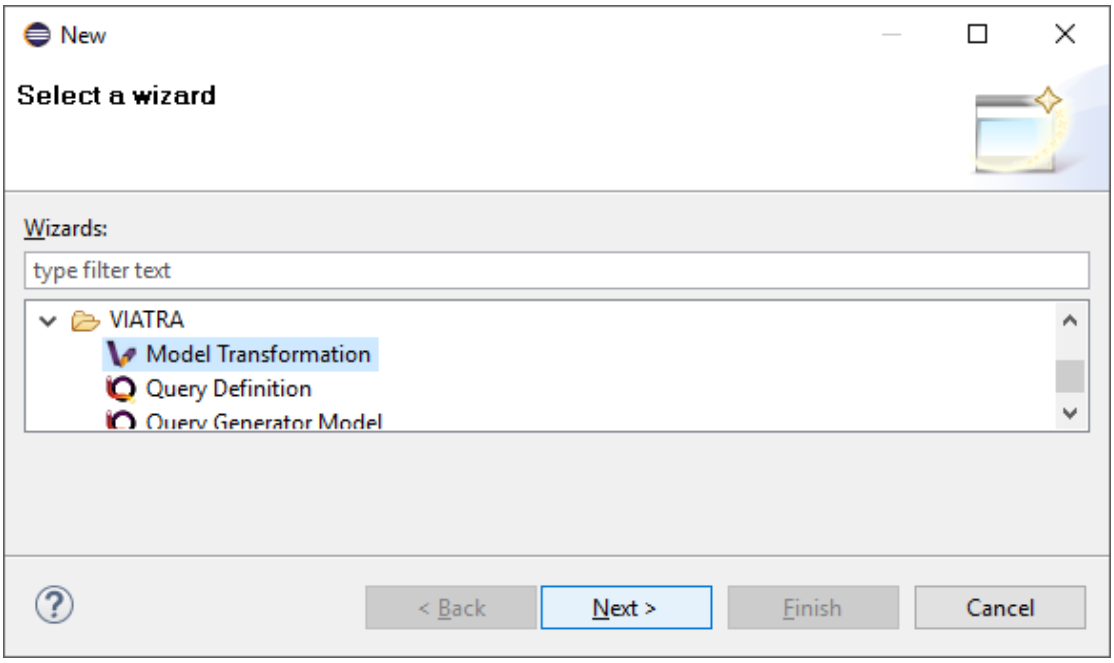

Figure 13. Model Transformation Wizard - Create new transformation

¬ Setup the name of the transformation and click Next

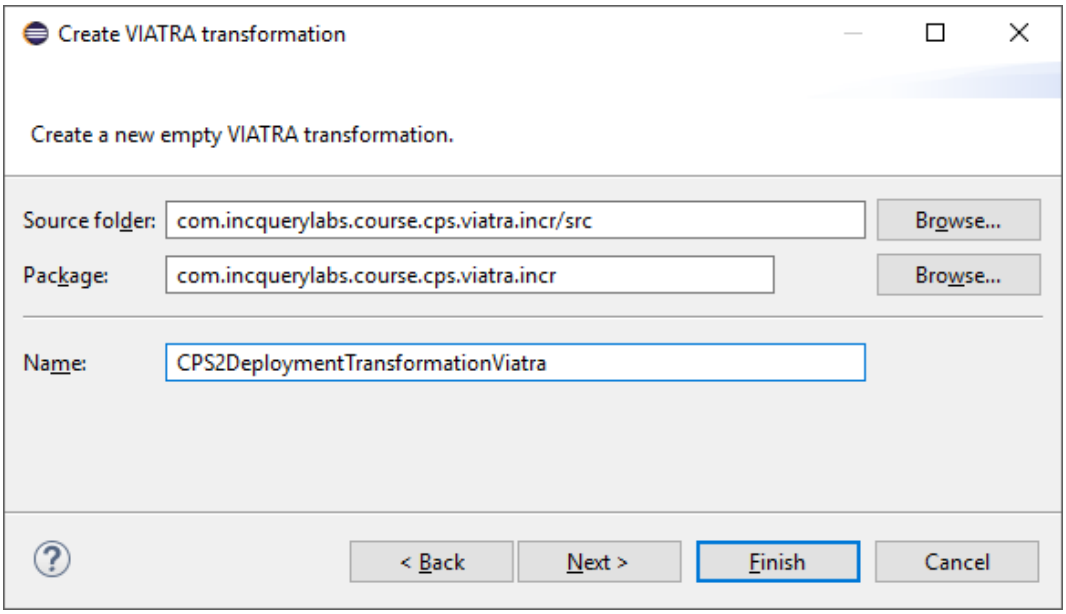

Figure 14. Model Transformation Wizard - Name of the new batch transformation

¬ Setup the type of the transformation to *BatchTransformation* and click Finish

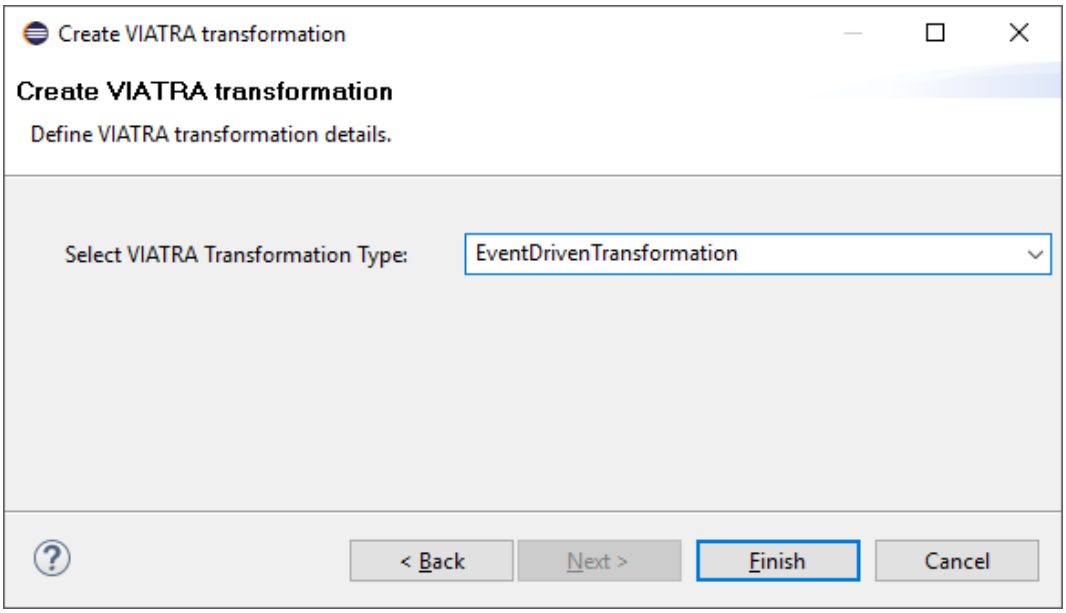

Figure 15. Model Transformation Wizard - Type of the new transformation

- Register used, domain-specific APIs as extensions, common APIs are already generated

```
/*
 * VIATRA Query group
 */
val extension CpsXformM2M cpsXformM2M = CpsXformM2M.instance

/*
 * EMF metamodels
 */
val extension DeploymentPackage depPackage =
DeploymentPackage::eINSTANCE
val extension TraceabilityPackage trPackage =
TraceabilityPackage::eINSTANCE
```

- Constructor will also initialize transformation (replace the generated one)
  - It assumes that the output and trace models are already created
  - The IModelManipulations implementation is used to make model access replaceable, this way the same transformation may be used for cases where the resource set is transactional

```
val CPSToDeployment cps2dep

new(CPSToDeployment cps2dep, ViatraQueryEngine engine) {
    this.cps2dep = cps2dep
    this.resource = cps2dep.deployment.eResource
    this.engine = engine
    prepare(engine)
    createTransformation
}
```

- Transformation will remain active until dispose is called (a `dispose` method is already generated)
- Create a rule to create DeploymentHosts for each HostInstances
  - The EventDrivenTransformationRuleFactory extension provides a builder API for rule definition
  - A VIATRA query pattern is used as precondition to the rule, which means the rule will be activated each time the given pattern changes allowing to update the output accordingly.

```
val hostRule =
createRule.precondition(HostInstanceMatcher.querySpecification)
```

- Add action for each kind of changes in the pattern to update trace and output models:
  - upon creation of a HostInstance

```
.action(CRUDActivationStateEnum.CREATED) [
    val hostinstance = hostInstance
    val nodeIp = hostInstance.nodeIp
    println('''Mapping host with IP: «nodeIp»''')
    /* Create new DeploymentHost element in output model */
    val host = cps2dep.deployment.createChild(deployment_Hosts,
deploymentHost) => [
        set(deploymentHost_Ip, nodeIp)
    ]
    /* Create trace entry */
    cps2dep.createChild(CPSToDeployment_Traces,
CPS2DeploymentTrace) => [
        addTo(CPS2DeploymentTrace_CpsElements, hostinstance)
        addTo(CPS2DeploymentTrace_DeploymentElements, host)
    ]
]
```

  - upon the change of a HostInstance

```
.action(CRUDActivationStateEnum.UPDATED) [
    /* find associated DeploymentHost element */
    val depHost = engine.cps2depTrace
                        .getOneArbitraryMatch(cps2dep, null,
hostInstance, null)
                        .depElement as DeploymentHost
    val hostIp = depHost.ip
    println('''Updating mapped host with IP: «hostIp»''')
    /* update IP attribute */
    val nodeIp = hostInstance.nodeIp
    depHost.set(deploymentHost_Ip, nodeIp)
    println('''Updated mapped host with IP: «nodeIp»''')
]
```

¬ upon the removal of a HostInstance

```
.action(CRUDActivationStateEnum.DELETED) [
    /* Find trace element */
    val traceMatch = engine.cps2depTrace
                        .getOneArbitraryMatch(cps2dep, null,
hostInstance, null)
    val hostIp = hostInstance.nodeIp
    println('''Removing host with IP: «hostIp»''')
    /* Remove DeploymentHost element */
    cps2dep.deployment.remove(deployment_Hosts,
traceMatch.depElement)
    /* Remove trace */
    cps2dep.remove(CPSToDeployment_Traces, traceMatch.trace)
    println('''Removed host with IP: «hostIp»''')
]
```

¬ Add default activation lifecycle then build the rule:

¬ The lifecycle defines the state machine used to determine the possible states on which transition actions can defined.

```
.addLifeCycle(Lifecycles.getDefault(true, true)).build
```

- The rule which create DeploymentApplication elements for ApplicationInstances, looks similar

```
val applicationRule =
createRule.precondition(ApplicationInstanceMatcher.querySpecifica
tion)
.action(CRUDActivationStateEnum.CREATED) [
    /* Find associated DeploymentHost for the HostInstance this
application is allocated to */
    val depHost =
engine.cps2depTrace.getAllValuesOfdepElement(null, null,
appInstance.allocatedTo).filter(
        DeploymentHost).head
    val appinstance = appInstance
    val appId = appInstance.identifier
    println('''Mapping application with ID: «appId»''')
    /* Create DeploymentApplication application in host */
    val app = depHost.createChild(deploymentHost_Applications,
deploymentApplication) => [
        set(deploymentApplication_Id, appId)
    ]
    /* create trace entry */
    cps2dep.createChild(CPSToDeployment_Traces,
CPS2DeploymentTrace) => [
        addTo(CPS2DeploymentTrace_CpsElements, appinstance)
        addTo(CPS2DeploymentTrace_DeploymentElements, app)
    ]
    println('''Mapped application with ID: «appId»''')
].action(CRUDActivationStateEnum.UPDATED) [
    /* find associated DeploymentApplication */
    val depApp =
engine.cps2depTrace.getOneArbitraryMatch(cps2dep, null,
appInstance, null).
        depElement as DeploymentApplication
    /* Update ID */
    if (depApp.id != appInstance.identifier)
        depApp.set(deploymentApplication_Id,
appInstance.identifier)
].action(CRUDActivationStateEnum.DELETED) [
    /* find associated DeploymentApplication */
    val trace = engine.cps2depTrace.getAllValuesOftrace(null,
appInstance, null).head as CPS2DeploymentTrace
    val depApp = trace.deploymentElements.head as
DeploymentApplication
    /* Remove application from host */

engine.allocatedDeploymentApplication.getAllValuesOfdepHost(depAp
p).head.remove(deploymentHost_Applications, depApp)
    /* Remove traces */
    cps2dep.remove(CPSToDeployment_Traces, trace)
].addLifeCycle(Lifecycles.getDefault(true, true)).build
```

- Replace the generated `createTransformation` using the rules defined above
  - For cases when it is possible to have more than one rules activated (e.g. a new HostInstance is added to the model with already set allocated applications) a conflict

resolver is used to provide a fixed ordering of rules to be executed.

¬ We use a priority-based resolver (lower priority rules will be executed first), which considers priority of disappearing rules to be inverted (a disappearing application's priority will be -2)

```
private def createTransformation() {
    //Initialize model manipulation API
    this.manipulation = new SimpleModelManipulations(engine)

    //Initialize event-driven transformation
    val fixedPriorityResolver = new
InvertedDisappearancePriorityConflictResolver

fixedPriorityResolver.setPriority(hostRule.ruleSpecification,
1)

fixedPriorityResolver.setPriority(applicationRule.ruleSpecifica
tion, 2)

    transformation =
EventDrivenTransformation.forEngine(engine)
        .setConflictResolver(fixedPriorityResolver)
        .addRule(hostRule)
        .addRule(applicationRule)
        .build
}
```

4.2.2. Creating a menu command to execute the transformation

- Create UI plugin
- Add dependencies:

  MANIFEST.MF

  ```
  org.eclipse.ui,
  com.incquerylabs.course.cps.viatra.incr;bundle-version="0.1.0",
  org.eclipse.viatra.examples.cps.traceability;bundle-
  version="0.1.0",
  org.eclipse.viatra.query.runtime;bundle-version="1.2.0"
  ```

- Create handler implementations:

ToggleTransformationHandler.java

```
public class ToggleTransformationHandler extends AbstractHandler
implements IHandler {

    ViatraQueryEngine engine;
    CPS2DeploymentTransformationViatra transformation;


    /* (non-Javadoc)
     * @see
org.eclipse.core.commands.IHandler#execute(org.eclipse.core.comma
nds.ExecutionEvent)
     */
    @Override
    public Object execute(ExecutionEvent event) throws
ExecutionException {
        IStructuredSelection selection =
            (IStructuredSelection)
HandlerUtil.getCurrentSelection(event);

        CPSToDeployment tracemodel =
            (CPSToDeployment) selection.getFirstElement();

        if(transformation == null) {
            if(engine == null) {
                try {
                    engine = ViatraQueryEngine.on(
                                new EMFScope(
                                    tracemodel.eResource()

.getResourceSet()));
                    transformation =
                        new
CPS2DeploymentTransformationViatra(tracemodel,

engine);
                } catch (ViatraQueryException e) {
                    throw new ExecutionException(e.getMessage(),
e);
                }
            }
        } else {
            transformation.dispose();
        }

        return null;
    }

}
```

- Register handler in the context menu of "CPSToDeployment" elements

plugin.xml

```
<extension point="org.eclipse.ui.commands">
    <command
defaultHandler="com.incquerylabs.course.cps.viatra.incr.ui.Toggle
TransformationHandler"

id="com.incquerylabs.course.cps.viatra.incr.ui.command"
            name="Toggle Transformation">
    </command>
</extension>
<extension point="org.eclipse.ui.menus">
    <menuContribution allPopups="false"

locationURI="popup:org.eclipse.ui.popup.any?after=additions">
        <command
commandId="com.incquerylabs.course.cps.viatra.incr.ui.command"
                label="Toggle Incremental Transformation"
                style="push">
            <visibleWhen checkEnabled="false">
                <with variable="selection">
                    <count value="1">
                    </count>
                    <iterate>
                        <adapt
type="org.eclipse.viatra.examples.cps.traceability.CPSToDeploymen
t">
                        </adapt>
                    </iterate>
                </with>
            </visibleWhen>
        </command>
    </menuContribution>
</extension>
```

4.2.3. Executing the transformation

- Launch runtime eclipse
- Create a generic resource project
- Copy a `.cyberphysicalsystem` resource in it

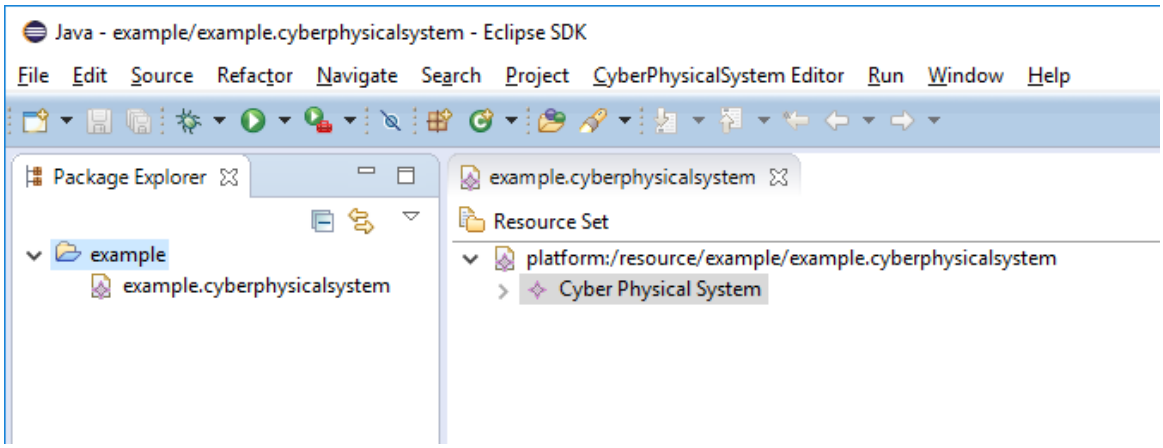

Figure 16. Project with a .cyberphysicalsystem resource

- Create an empty Deployment model
  - Root element shall be Deployment

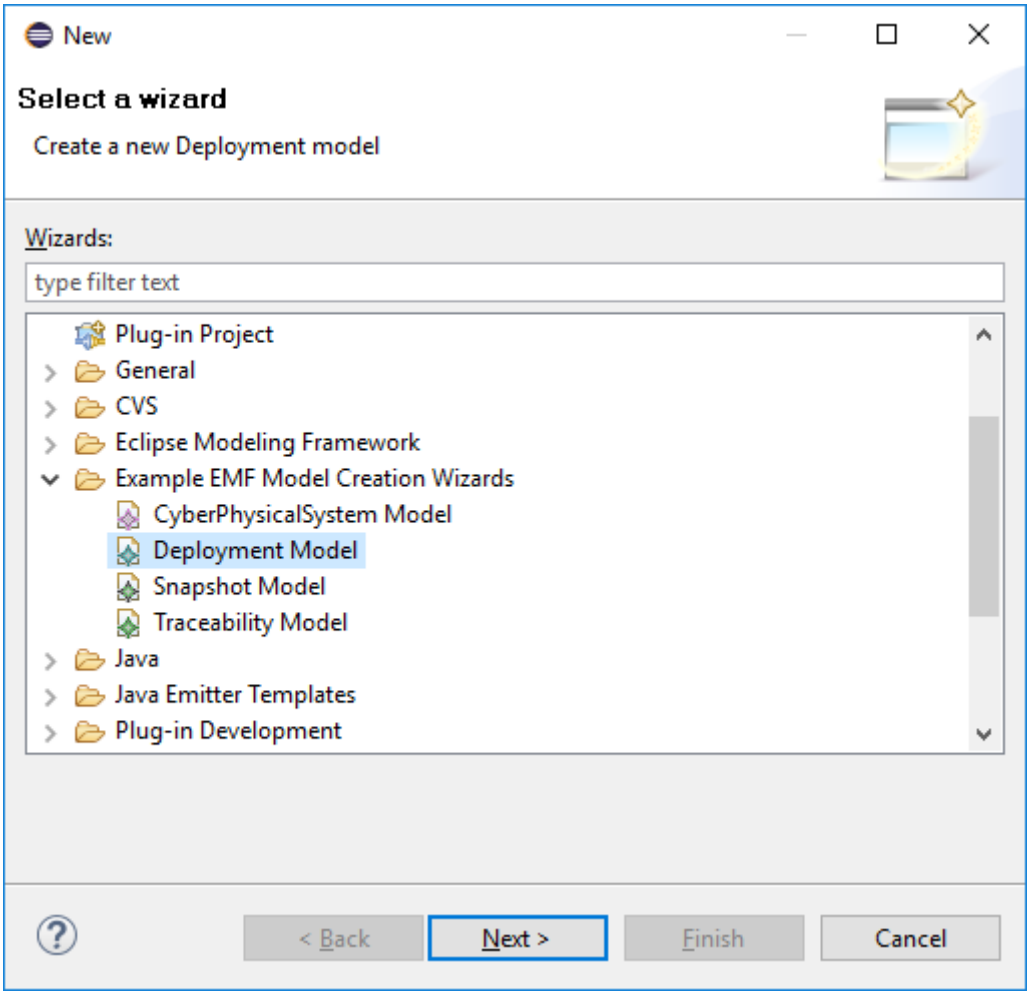

Figure 17. New Deployment Model

- Create a Traceability model
  - Root element shall be "CPS To Deployment"

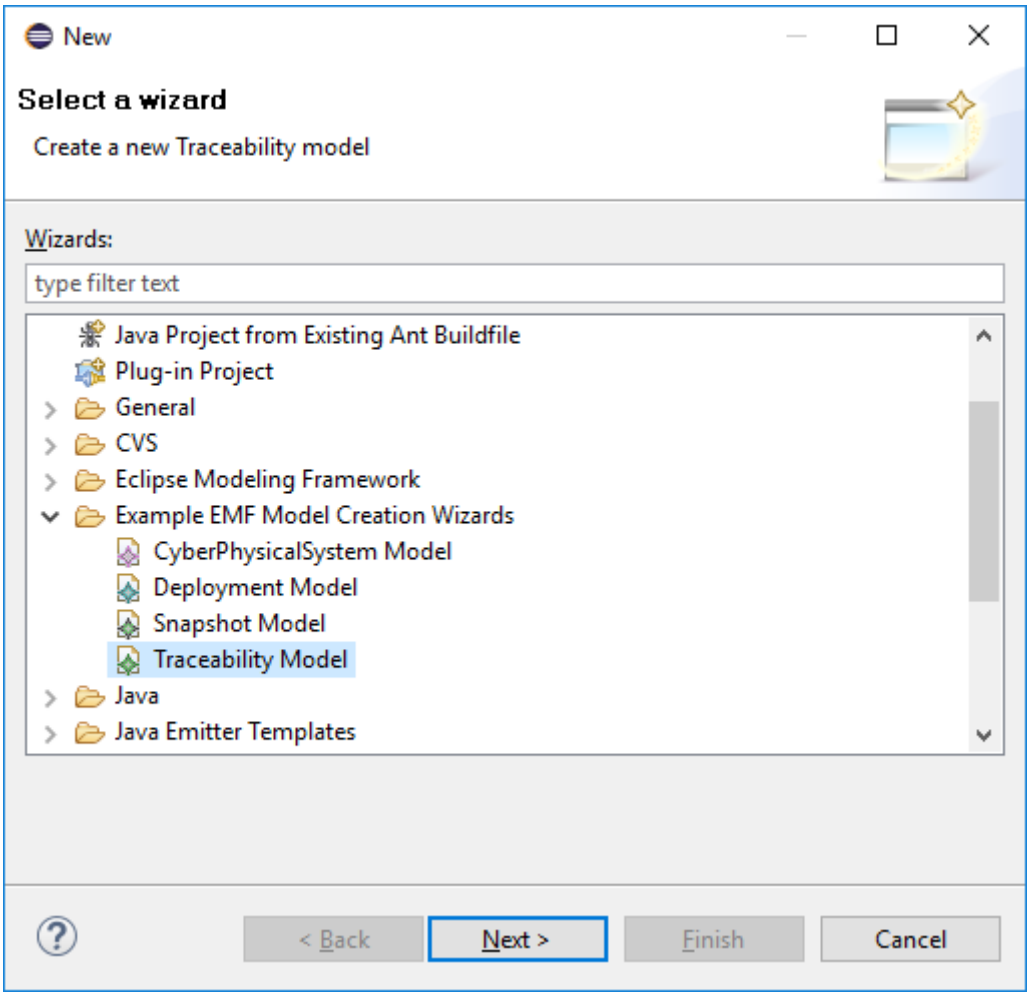

Figure 18. New Traceability Model

- In the Traceability editor, load both CPS and Deployment models with "Load Resources.." in the context menu

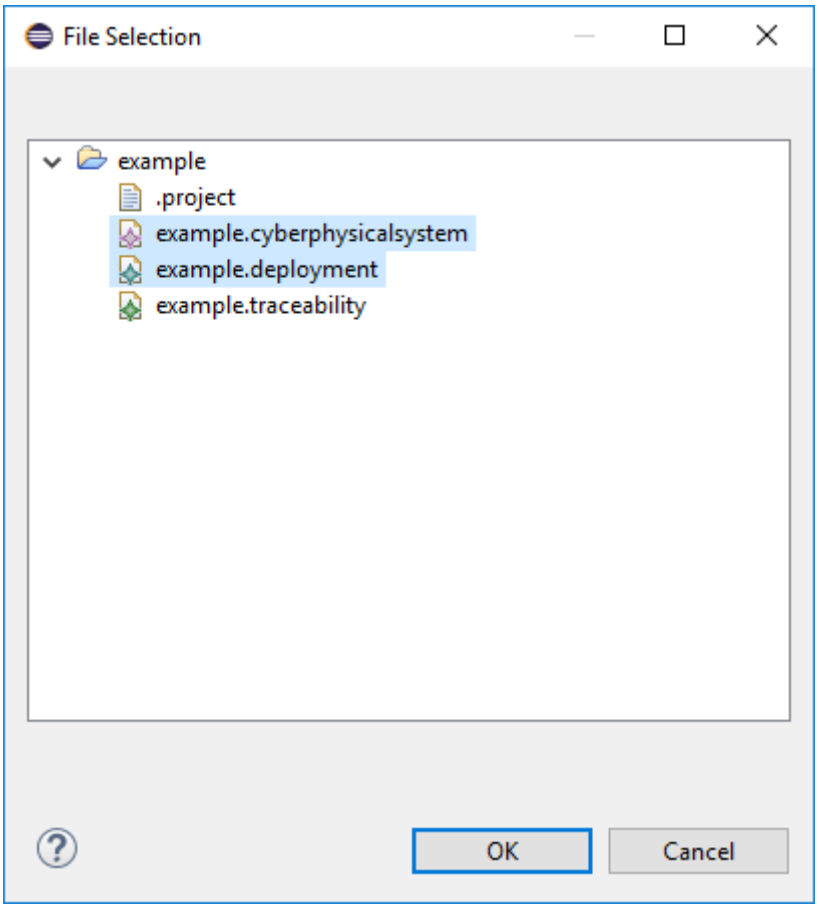

Figure 19. Load necessary resources into the Traceability Model

- Set CPS and Deployment references of traceability model in the properties view

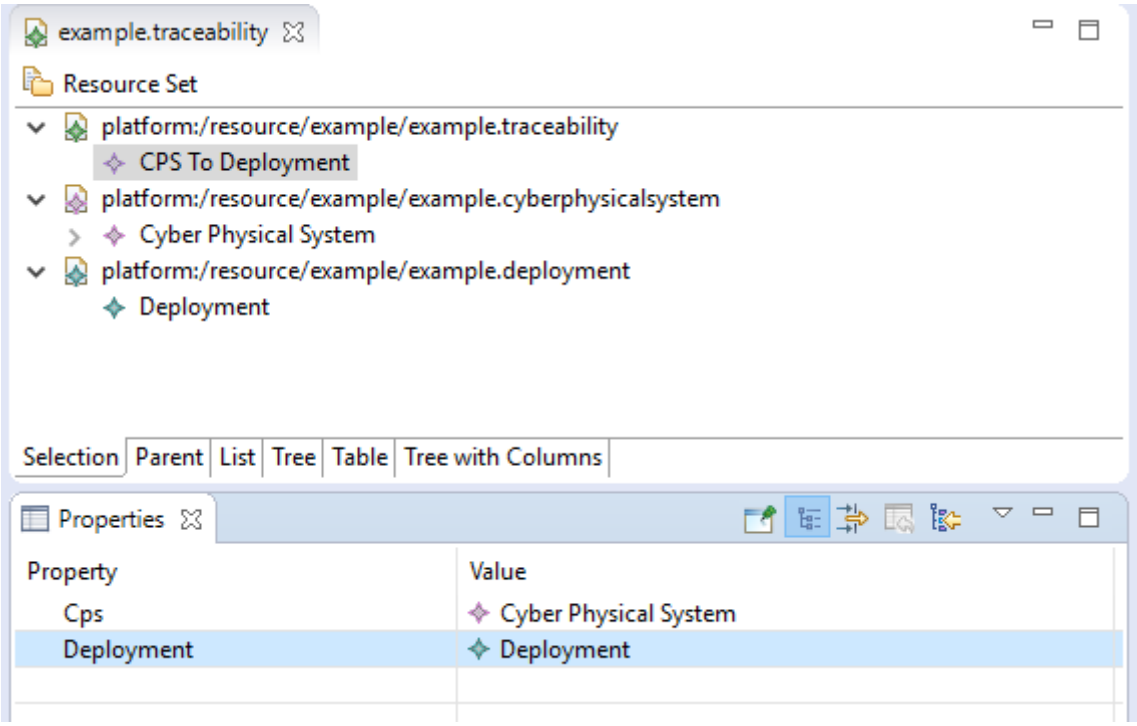

Figure 20. Set the references of the Traceability Model

- Toggle transformation using the created command (on the context menu of the

Traceability model root)

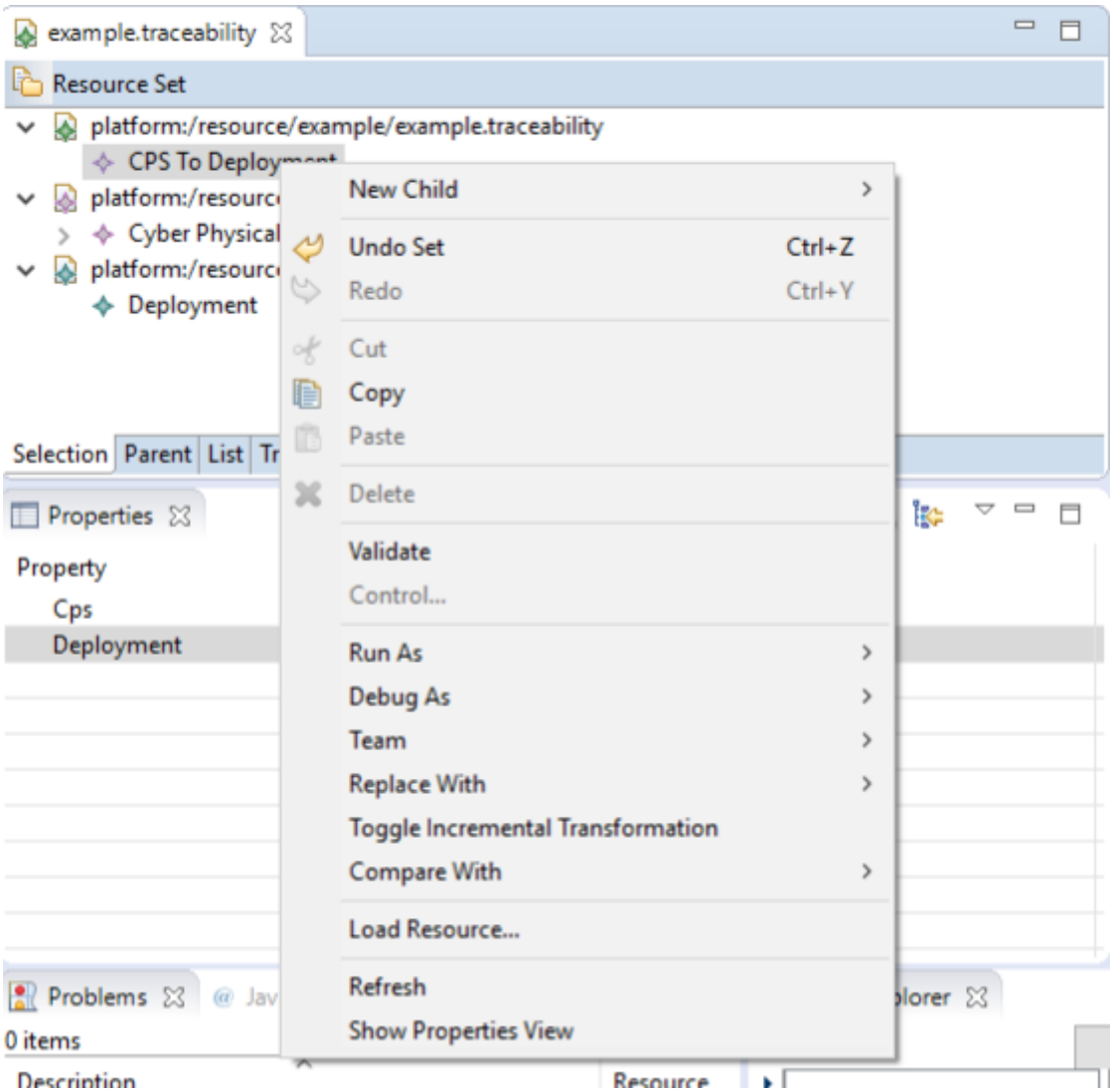

Figure 21. Toggle transformation in the context menu

- Initial activation done on first modification of the input model, e.g. create a new HostType

4.2.4. References

- VIATRA Transformation API: https://wiki.eclipse.org/¬VIATRA/¬Transformation/¬Transformation_API
- Further transformation examples: https://wiki.eclipse.org/¬VIATRA/¬Transformation/¬Examples